ISI/TM-78-7
January 1978
ARPA ORDER NO. 2223

LEVEL II

PRIM System:

- Tool Builders Manual
- User Reference Manual

Louis Gallenson
Alvin Cooperband
Joel Goldberg

INFORMATION SCIENCES INSTITUTE
UNIVERSITY OF SOUTHERN CALIFORNIA
4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER (14) ISI/TM-78-7	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* (6) PRIM System: PRIM Tool Builders Manual and User Reference Manual.	5. TYPE OF REPORT & PERIOD COVERED (9) Technical manual.
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) Louis Gallenson, Alvin Cooperband, Joel Goldberg	8. CONTRACT OR GRANT NUMBER(s) (15) DAHC 15-72-C-0308
9. PERFORMING ORGANIZATION NAME AND ADDRESS Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90291	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: ARPA Order-2223
11. CONTROLLING OFFICE NAME AND ADDRESS (11) Defense Advanced Research Projects Agency, 1400 Wilson Blvd., Arlington, VA 22209	12. REPORT DATE: January 1978 13. NUMBER OF PAGES: 121
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)* (12) 154p.	15. SECURITY CLASS. *(of this report)*: Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

This document approved for public release and sale; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

debugging tool, emulated I/O, emulation-based programming tools, emulators, microprogramming

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This is a two-part manual for developers of PRIM-based emulators. The manual describes the capabilities of PRIM, the requirements for PRIM-based emulators, and the MLP-900 microprogrammable processor.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED (SB)
SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

407 952

ISI/TM-78-7
January 1978

ARPA ORDER NO. 2223

PRIM System:

- Tool Builders Manual
- User Reference Manual

Louis Gallenson
Alvin Cooperband
Joel Goldberg

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

THIS RESEARCH IS SUPPORTED BY THE ADVANCED RESEARCH PROJECTS AGENCY UNDER CONTRACT NO. DAHC15 72 C 0308, ARPA ORDER NO. 2223, PROGRAM CODE NO. 3D30 AND 3P10.

PRIM Tool Builders Manual*

Abstract

PRIM is an interactive microprogrammable environment used for creating and running emulators of existing or newly specified computers, with major emphasis on debugging tools that can be operated by the user in the language of the original system. This document serves as a manual for programmers interested in writing emulation tools to run under PRIM. It covers an overview of the PRIM system design, requirements for emulators that are to run under a PRIM framework, the MLP-900 microprogrammable processor, the GPM language for programming emulators to run on the MLP-900, PRIM exec and debugger commands for the tool builder that supplement the commands available to the general PRIM user, and the TENEX MLP-900 driver interface for those MLP-900 users who might not want to run under PRIM.

---

* This research is supported by the Advanced Research Projects Agency under Contract No. DAHC15 72 C 0308, ARPA Order No. 2223.

Contents

Preface

This manual is intended for PRIM users who are interested in building their own emulation tools or in extensively modifying existing tools. The tool builder must be aware of three levels of control or interface protocols. At the first level, the PRIM framework must interface to the operating system in which it is is embedded (TENEX or NSW). To utilize the PRIM system, the tool builder--as well as the tool user--must have access to and knowledge of the basic commands of the appropriate operating system (operating manuals for TENEX and NSW are generally available to interested users; information contained in such manuals is beyond the scope of this manual). A second level of interfacing is between the user and the PRIM exec or debugger command interpreters (general user information for PRIM can be found in ***PRIM System: Overview*** and ***PRIM System: User Reference Manual***, which tool builders are assumed to have read; information for specific existing tools can be found in a ***User Guide*** for that tool). The third level of interface, between an emulator and the PRIM framework, is covered in this manual.

The manual is organized into four chapters and three appendices. Chapter 1 presents an overview of the PRIM system design. Chapter 2 discusses requirements for emulators that are to run under a PRIM framework. Chapter 3 describes the MLP-900 microprogrammable processor. Chapter 4 presents the GPM language for programming emulators to run on the MLP-900. Appendix A discusses those PRIM exec and debugger commands available to the tool builder that supplement the commands available to the general PRIM user. Appendix B describes the TENEX MLP-900 driver interface for those MLP-900 users who might not want to run under PRIM. Appendix C lists GPM reserved words.

Chapter 1
Overview of the PRIM System Design

For some applications the native machine is not the system of choice in which to develop software, as when the target machine is unavailable (because it is still being developed, is obsolete, or is inaccessible) or inconvenient (as when there is minimal target-system support for debugging). In such cases, simulation or emulation may be preferred. Simulation has the advantage of giving the user intimate access to the target machine, usually through a rich debugging package. Typically, however, this richness is achieved at a high development cost for the simulator and at a target-system performance degradation of four or more orders of magnitude. Emulation can offer processing speeds comparable to the target system (even faster, for slow target machines), but typically does not support a rich debugging environment. The PRIM system attempts to retain the best features of both the simulation and emulation approaches while at the same time minimizing their disadvantages. PRIM provides a sharable, uniform framework for running emulations of target machines; within that framework is a rich user interface that supports interactive target-system and emulator debugging. When the user is not engaged in debugging, the target system runs at emulator speeds, but a sophisticated debugging package is available immediately when needed. PRIM was developed within the TENEX timesharing system so as to provide convenient access, a file system, resource management, and a large set of utilities without the cost of developing yet another operating system.

By cleanly and sharply separating the debugging and target-machine emulation tasks, PRIM has been able to avoid most of the disadvantages of simulation and emulation while at the same time combining their advantages. In achieving this sharp separation of function, PRIM established a uniform and systematic structure for the development of emulators. This structure not only minimizes the involvement of the emulator in the debugging process, but also greatly simplifies the task of emulator development as it utilizes a standard package of I/O service routines and provides a convenient control structure suitable for a large family of target-machine emulations. As most of PRIM consists of sharable system-level and user-level code that is common to this potentially large family of target system emulations, a more extensive development effort (with its consequently more sophisticated design) was called for than would have been appropriate for a single-machine emulation or simulation.

1.1 The PRIM System Architecture

The emulation of a target machine under PRIM involves three different system levels: the TENEX timesharing system, which runs on a PDP-10; the PRIM framework, which runs at user level under TENEX; and target-machine emulation tools controlled by that framework, which run on a sharable MLP-900 microprogrammable processor. The timesharing system hardware and software provide shared access to the MLP-900. The PRIM framework supports interactive users at terminals and provides access to the file system for the emulator. The emulator maintains the complete target-system environment. The PRIM framework can be used for both emulator development and target program debugging.

The PDP-10 is a large, general purpose computer to which new devices can be connected fairly easily, since the I/O bus is extensible and the multiported memory is external to the processor. TENEX is strongly oriented toward the support of interactive computing, serving both local users and remote users connected via the ARPANET. It does not interact directly

with the user, but rather allocates resources, manages the file system, and supports the execution of TENEX processes, each process running in its own paged virtual memory and interacting as appropriate with its own user via a terminal of some kind. To support PRIM, TENEX was extended with software to provide access to the MLP-900 by TENEX processes; the MLP-900 was extended with hardware and software to guarantee the integrity of TENEX, even against errant microcode.

The MLP-900 is a large, fast, vertical-word, microprogrammable processor with a writable control memory. The processor consists of an operating engine and a control engine. The operating engine is a 36-bit arithmetic/shift unit with 32 general registers, 16 mask registers, and a 1K internal memory. The control engine is a control unit with interrupt and branch logic, a subroutine-call stack, 128 programmable flip/flops, and 4K of writable control memory. Cycle time is 300 nanoseconds, during which either one or both engines can execute a 32-bit instruction. The MLP-900 is interfaced as a peripheral device on the PDP-10 I/O bus with direct access to the PDP-10 memory via one of the four existing memory ports; it has no peripheral devices of its own. The I/O bus interface allows the exchange of control information between the MLP-900 and the PDP-10; via this interface, either processor can interrupt the other. Hardware modifications were required only in the MLP-900; they consisted of the interfaces to the PDP-10 and a supervisor/user state that provides protection against user microcode for the I/O bus interface, the MLP pager (an address translator in the memory interface that mimics the TENEX pager), most of the MLP-900 interrupt system, and the MLP-900 control memory itself.

At the system level the software consists of a small operating system resident in the MLP-900, known as the microvisor, and a TENEX device driver to shake hands with the microvisor and govern access to the MLP-900 by TENEX processes. The MLP device driver is the only module added to the TENEX operating system; it allows a TENEX process to create, run, and control a subordinate MLP process in much the same way it can a subordinate TENEX process. It also schedules use of the MLP-900 among contending users and supervises the microvisor. Most of the microvisor is devoted to swapping emulator contexts (control memory and MLP registers) as the driver passes control of the MLP-900 from one user to another; the rest responds to emulator requests for service, manages the MLP pager, and performs other tasks required by the driver in TENEX. The microvisor runs in the privileged supervisor state that allows access to all resources; emulator microcode runs in the user state that protects all the critical resources from modification. PDP-10 memory is not directly addressed by microcode; instead, memory references are to addresses in a virtual memory identical to that of a TENEX process. These virtual addresses are translated to real addresses by the MLP pager, which is controlled (via the microvisor) by the driver in TENEX. A reference to a page not in memory results in a page-fault interrupt into the microvisor, which passes the fault to the driver and retries the memory operation after the page is fetched by TENEX.

The net effect of this design is a sharable emulation facility in which each emulator runs independent of all others in its own context, accessing its own virtual memory under control of the PRIM framework that created it. The framework has potential access to all of its emulator's context and memory and may inspect and/or modify them.

1.2 The PRIM Framework

The PRIM framework consists of TENEX processes that define and implement the PRIM user command language, create an MLP-900 emulation process and control its execution at the user's behest, and provide I/O service for that emulation process. The I/O service implements

a set of primitives that allow the emulator to transfer data to or from the TENEX file system. The emulator invokes these primitives to perform target I/O operations on installed devices after the user has associated them with TENEX files.

An emulator is required to cooperate in the debugging process, although the demands are minimal. Essentially, an emulator is expected to stop cleanly when interrupted by the framework or on the occurrence of a small number of predefined events that it monitors and to report its reason(s) for stopping. When the emulator halts or is interrupted by user intervention, control returns to the user at command level via the framework.

The tool builder must supply the PRIM framework with tables that define the target system architecure and symbols and drive a target system assembler and disassembler. Except for machine and user symbols and target assembly language, the same command interactions apply to the use of every emulator and framework. The framework contains two separate command processors, known as the exec and the debugger. Although both offer automatic command completion and help facilities, each uses a language tailored to its functions. Typically, a user interacts with the exec only briefly when starting and ending a session; during the session he interacts primarily with a target program or the debugger. General user-level exec and debugger commands are discussed in detail in *PRIM System: User Reference Manual*; additional commands available to emulator developers are presented in Appendix A.

1.3 The PRIM Emulation Model

A prototypical PRIM emulator has been developed based on the constraints of the emulator's environment, the objectives of the PRIM system, the requirements of the PRIM framework, and the specific interface conventions that framework defines. The environment consists of execute-only microcode residing in control memory, the MLP-900 registers, and a 256K 36-bit (virtual) main memory; the registers and virtual memory together comprise the context into which are mapped the registers and memory of the target machine plus various other regions devoted to required PRIM functions. The mapping is arranged at the convenience of the tool builder, with accompanying tables describing this mapping to the PRIM framework. The emulator can modify its context in the course of emulation, stop (thereby returning control to the framework), and request I/O services from the framework.

The prototypical control structure allows an emulator to stop after any cycle and subsequently resume emulation in a manner totally transparent to the target machine. While a single target instruction constitutes the typical cycle, other events, such as interrupts or I/O data transfers, are also treated as emulator cycles.

Target timing in PRIM is virtual. The prototypical emulator increments an internal, high-resolution, virtual timer to reflect the passage of target-machine time; there is no fixed relationship among target time, MLP-900 time, PRIM framework time, and real time. Emulated cycles that consume target time (*e.g.*, instruction execution) advance the virtual timer; emulated cycles that nominally occur in parallel with the former (*e.g.*, I/O controller activity) are scheduled for service relative to that timer. The result is a small, event-driven, discrete simulation system with target instruction execution being treated as a background task.

Chapter 2
Emulation Tool Requirements

This chapter presents emulation tool requirements in eight parts: (1) the microvisor environment, (2) the framework environment, (3) I/O emulation and timing, (4) I/O server, (5) breakpointing, (6) overall control structure, (7) tool descriptor tables, and (8) emulator installation.

2.1 The Microvisor Environment

The MLP-900 microprogrammable processor is a sharable resource of the TENEX system. Access is controlled by the TENEX MLP driver, which together with the MLP microvisor allows TENEX processes to run emulation processes in a time-shared MLP-900. There is no interaction among emulation processes.

An emulator consists of microcode written in the GPM language to run in user state on the MLP-900 (see Chapters 3 and 4). Co-resident with the emulator in MLP-900 control memory, occupying locations 7000 through 7755 octal, is the microvisor--the operating system under which the emulator runs. The emulator resides in the remainder of control memory and has available all the nonprivileged registers of the MLP-900. These control memory locations and registers together constitute the emulator's context; all of it is swapped into the MLP-900 when the emulator is started and all but control memory is swapped out when the emulator is stopped. The emulator can read and write a (virtual) main memory of 256K 36-bit words.

The microvisor runs in supervisor state. It processes all privileged action requests and provides a set of routines that can be called by an emulator to perform necessary services. The entry points for these calls are defined in the file <PRIM>ENTRY-VECTOR.GPM, which should be included in every emulator (see the GPM INCLUDE command in Section 4.4.3).

2.1.1 Action Requests

The servicing of privileged action requests (AR's) by the microvisor is completely transparent to the emulator. The principal such services concern the swapping of emulator contexts into and out of the MLP-900 and the servicing of page faults that occur on emulator references to main memory.

There are eight user-level action requests (see Section 3.3.3), flops F.130 through F.137, governed by the flop ARL.5 (F.150). Associated with these user AR's are the interrupt locations 7756 through 7775 octal, respectively, which are considered part of user control memory. The TENEX MLP-900 driver and the microvisor cooperate to permit the controlling TENEX process to set any of these user AR's while the emulator is running; unless ARL.5 is set, a user-level AR interrupt results.

2.1.2 Extended Stack

The microvisor supports an extended stack that is used when hardware-stack overflow or underflow occurs. Whenever the microvisor is entered, whether by a call or an AR interrupt, two levels of the stack are used. As a result, an emulator may require the extended stack if it uses more than twelve levels of routine nesting, including nesting due to user-level AR interrupts.

The extended stack consists of sixteen 16-word blocks (in each of which the first word and last two words are not touched) in the last page of MLP-900 auxiliary memory (locations A.1400 through A.1777). The total capacity of the extended stack is 210 words. The tool builder may treat as general auxiliary storage those blocks not needed for the extended stack. The four high-order bits of P.6 select the block to use when the hardware stack overflows. When the extended stack itself overflows, block zero is used. Incrementing or decrementing P.6 through zero produces an extended-stack overflow and causes an emulator error-stop. It should be noted that since not every word of a block of the extended stack is used, P.6 may not go up and down uniformly by one on calls and returns.

2.1.3 Stopping and Starting

Whenever the controlling TENEX process runs/resumes an emulator, the microvisor returns control via the top of the stack. An emulator is stopped, its context swapped out of the MLP-900, and the controlling TENEX process notified on any of the following:

- The emulator calls MLP.STOP
- The controlling TENEX process halts the emulator.
- Any of the following action requests occur: CMADR, SUPVF, PROT, or VADR.
- An extended-stack overflow or underflow occurs.
- A reference is made to a protected page of memory.

2.1.4 Microvisor Calls

The available entry points and calling sequences for microvisor routines are:

- MLP.STOP - stop until resumed externally. The calling sequence is

```
CALL MLP.STOP ;
```

  A call to MLP.STOP relinquishes control of the MLP-900. The microvisor and TENEX MLP driver swap the emulator's context out of the MLP and notify the controlling TENEX process of the stop. If that process resumes emulation, the call to MLP.STOP returns at the next micro-instruction.

- MLP.CALL - pass parameter to the controlling TENEX process. The calling sequence is

```
R.37 ← call parameter ;
CALL MLP.CALL ;
```

  The parameter value in R.37 is passed to the controlling TENEX process via the TENEX MLP driver. (The PRIM framework interprets calls to MLP.CALL as requests for I/O service; see Section 2.4.) After the parameter word is passed to the driver, the call to MLP.CALL returns at the next micro-instruction.

- MLP.RCM - read control memory. The calling sequence is

```
(P.2) ← control memory address ;
CALL MLP.RCM ;
```

A call to MLP.RCM allows user microcode to examine MLP-900 control memory. The call returns immediately to the next micro-instruction with the contents of the designated control memory location in R.37.

2.2 The Framework Environment

PRIM is intended to support emulations of small- to medium-sized computers, with word sizes up to 32 bits and I/O configurations of moderate size and variety, including tapes, disks, terminals, and unit-record equipment. The tool builder should strive for a complete, bit-compatible emulation of the target machine, including not just instructions and registers, but also clocks, interrupts, machine states, memory protection and relocation, and nearly real I/O.

2.2.1 Required Allocations

The PRIM framework requires main memory to be divided into three fixed regions: working memory, buffer memory, and configuration memory. Buffer memory is defined in the emulator's descriptor tables by *buflow* and *bufhi* (see Section 2.7.2). The region below *buflow* is working memory, containing target memory and any other large storage areas needed by the emulator. Buffer memory is shared between the I/O server and the emulator for transfers to and from the TENEX file system. Configuration memory lies above *bufhi* and contains machine and device parameters; at a minimum it consists of the last page of main memory, addresses 777000 through 777777 octal. Each device parameter must be allocated within its device control block (see Section 2.3.2); global machine parameters may reside either in MLP-900 auxiliary memory or in configuration memory.

A few flops associated with MLP-900 action-request interrupts are used by PRIM for fixed purposes.

- F.130 (TRAC) and F.153 (ITRAC) are used in the implementation of the MLP-single-step command, as are the two control memory locations associated with the TRAC action request, 7756 and 7757 octal. The emulator should avoid using any of these locations.
- F.131 is the STATUS action request, requiring the emulator to return the target status to the framework as soon as possible. The emulator may report status either by stopping--in which case STATUS and QUIT are identical--or by issuing an RSTAT call (see Section 2.4.2.1).
- F.132 is the QUIT action request, requiring the emulator to stop at the end of the current cycle (see Section 2.6.3).

The emulator must either contain interrupt routines to handle F.131 and F.132 interrupts or disable user interrupts and poll F.131 and F.132.

2.2.2 Mainframe Emulation Standards

A complete instruction set, functionally identical to that in the emulated CPU, must be implemented. As the actual details of implementation are transparent to the user, the emulation need be verifiable only at the point where the emulator stops. The possibility of additional meta-instructions for any given machine should not be precluded.

A full interrupt facility, functionally identical to that in the emulated CPU, must be implemented. For interrupt conditions that cannot occur because execution is emulated (*e.g.*, a

memory parity error), there should still be some way for the user to cause the interrupt, such as implementing a user-settable indicator that represents the pending interrupt condition. In general, the detection of an interrupt condition and the "taking" of the interrupt are best treated as distinct emulator cycles with a set of interrupt-pending bits holding the state information between the two cycles.

2.2.3 Registers and Switches

All programmable registers and switches should be implemented, except those listed below. Momentary switches (*e.g.*, master-clear/reset) are to be noticed by the emulator and cleared; normal toggles are to be read-only. It is not necessary to store console toggles in MLP flops; they can be assigned where most convenient. The following switches should not be implemented:

- Power On/Off
- Run/Stop switch (it is replaced by the external control of the emulator itself).
- Switches used for machine diagnostics and maintenance that deal with machine minor cycles and such nasty things.

2.2.4 Target Memory

Target memory is mapped into the emulator's working (PDP-10) memory, beginning at location zero and using as many words as required. The packing of target words is restricted only in that the four high-order bits of each 36-bit PDP-10 word are reserved for meta-bits. In general, it is recommended that a single addressable target memory location be stored in each PDP-10 word.

Any memory paging, relocation, and protection offered by the target machine must be implemented faithfully by the emulator at the functional level. Target memory is to be treated as the machine's physical memory, not as a target user's virtual memory.

2.2.5 Timing and Emulated Clocks

Timing is done through an internal (36-bit) virtual timer whose unit is some smallest time of interest, at most a machine minor cycle. The interval must be fine enough for accurate timing; it must be large enough that the number of intervals required to schedule the longest event can be represented in 35 bits. A time of 50 nanoseconds is proposed as a standard, allowing events of approximately 30 minutes duration. Instruction execution, memory references (by devices or CPU), and anything else that takes time, all advance the virtual timer appropriately. For machines with inexact timing--due to asynchronous functional units, interleaved memory, or data-dependent execution times--only statistically correct timing may be possible.

The virtual timer is used not only to establish emulated execution time but also to provide a time frame in which to run I/O devices, slow clocks, and whatever else operates (infrequently) in parallel with the mainframe. This time frame allows the handlers for such devices to schedule themselves for service at (regular or irregular) intervals to reflect asynchronous operation accurately. The emulator treats the virtual timer as a continuous circular counter; PRIM keeps track of the high order portion for purposes of reporting time to the user.

2.2.6 Time Synchronization

Synchronization of emulated (virtual) time with the PRIM framework--and, through it, with other processes or emulations--is an optional feature. If implemented, it requires a global parameter that will contain the synchronization interval and a scheduled pseudo-clock that generates an RSTAT call (see Section 2.4.2.1) at the end of each such interval of virtual time.

2.3 I/O Emulation

All I/O device control and timing is emulated. Each device type supported by an emulator is implemented by a microcoded device handler. Execution of a device handler is scheduled relative to the (high-resolution) virtual timer: the initial execution of a handler is scheduled by the CPU or by emulator initialization, then each time a handler runs it reschedules itself for its next execution as necessary.

The device handler is responsible for all of the control and state logic associated with the emulated device. The data medium of the device is the TENEX file system; the PRIM framework includes an I/O server that gives the device handler access to it.

2.3.1 I/O Configuration

Configuration in PRIM consists of the user "installing" supported I/O devices, "mounting" files on installed devices, and specifying assorted parameters associated with these devices and files. The installation of devices is allowed only prior to initializing the emulator and may, therefore, be assumed to be fixed over a normal emulator stop/resume sequence. Although the mounting of files is dynamic, with the user able to change file assignments and characteristics at any time, it is of no concern to the emulator, as the I/O server is responsible for all TENEX file management. Device parameters are divided into two classes: those that may be set only at installation time and those that may be altered by the user any time the emulator is stopped. Device parameters (and the class to which they belong) are defined in the emulator's descriptor tables (see Section 2.7.11; parameters marked EXPLICIT or FIXED may be set only at installation time, and those marked DEFAULT may be modified at any time).

Up to 64 devices may be installed. Each installed device is assigned a PRIM device number (PDN) by the framework as it is installed. A given device type may be installed any number of times, but each instance must have a unique device address (see the discussion for word 5 of a device control block in Section 2.3.2).

2.3.2 Device Slots

The PRIM framework contains 64 identical, configurable device slots, each capable of handling the I/O requirements of one emulated device. There is a one-to-one correspondence between device slot and PDN. The actual assignment of device slots to emulated devices is a configuration function. Associated with each of the 64 device slots is an eight-word device control block in configuration memory (at octal location 777xx0, where xx is the PDN). Each block is for the exclusive use of its device and handler; it includes both the (fixed) configuration information that is passed to the emulator and the I/O call block for executing actual I/O operations. The allocation of control block words is given below:

0. I/O-server call-block code and status word (initially zero).

1-3. I/O-server call-block parameters (not initialized).

4. Device handler parameters and private storage. This word contains device-specific parameters; any parts not used for parameters are initially zero.

5. Handler type (H.0--bits B4-19) and a unique device-address (H.1--bits B20-35). The handler type identifies the device handler for this device in this emulator and thus associates an emulated handler type with this control block. A handler type of zero (actually a word of zero) is reserved to mark an unused control block; handler indices thus begin with one. The device address is the location in the target-system I/O address space for this device and thus associates a target device with its control block. The device address will usually consist of an 8-bit primary (channel and/or controller) address and, where needed, an 8-bit secondary (unit) address.

6. <XWD *<buffer size (in words)>*, *<buffer first word address>*>. For devices that do not require a buffer (where all I/O is done with BIN or BOUT calls) this word is zero. Since buffer size requirements for each device type are determined at configuration time from the emulator's descriptor tables, all buffers will be the right size.

7. Device time. This is a 36-bit value giving some basic time unit for the device (in units of the virtual timer interval), typically the inter-byte transfer time. It should be used to pace the emulated device properly. The user should be able speed up or slow down a device by altering this value.

2.3.3 Device Handlers

Each different type of device (or controller) is implemented by a microcoded device handler that issues the necessary calls to the I/O server in the PRIM framework. The creation of new handlers for a given emulator is a demand function; the possible number of handlers is limited principally by the size of control memory. Each handler must be made known to the framework via appropriate descriptor-table information to allow the proper installation of the implemented device. Each installed device is associated with a unique device control block; the PRIM I/O server transfers data between the TENEX file system and that control block or its buffer in response to emulator calls.

The emulator performs a target I/O command by locating the appropriate device control block and using the handler type it contains to select the proper handler. A device handler must implement the I/O operations relevant to its device, using I/O server calls to manipulate the associated TENEX files. It must move transferred data between the appropriate mainframe locations and the allocated device buffer (or the control block, for single-byte transfers). It must also emulate the device timing, using its device time parameter to schedule its next execution. A device handler must be re-entrant so that a device or controller can be installed more than once; thus all storage required by a device handler (including the device control block) must be associated with a device slot and not with the device handler itself.

The level of detail in the I/O emulation is determined by the requirements of the expected applications. Thus for some applications a card reader that reads in the entire card

at once (at the conclusion of the necessary elapsed emulated time) might be adequate, while other applications might require a card reader to schedule the read of each column separately.

The internal structure of device handlers, and the conventions for interacting with the mainframe, are not here specified (except implicitly by the configuration requirement that all handlers work indirectly through control blocks). Certain large classes of emulated machines (*e.g.*, NTDS) will use common programming conventions for all their handlers in order to share common devices.

2.4 I/O Server

I/O service is performed by the PRIM framework asynchronously and in parallel with emulator execution. I/O transfers take place between the TENEX file system and device buffers (for multi-byte transfers) or device control blocks (for single-byte transfers). A device control block (containing a four-word call block) is set aside in configuration memory by the framework as part of device slot allocation (see Section 2.3.2), but a fixed relationship between call block addresses and particular devices is neither required nor assumed. The emulator issues a service request by building a call block and passing its address to the framework:

```
R.37 ← call block address ! the high-order half must be zero
CALL MLP.CALL ;
```

The I/O server performs the requested operation using files currently mounted on the designated device, replying (and returning status) in the call block itself. When an operation completes, the server sets a call-completed bit in the call block. Any number of requests may be outstanding simultaneously, but only one call may be outstanding at a time from any given call block.

2.4.1 Supported Device Classes

The I/O server supports three classes of emulated devices: sequential (communications, paper tape, and unit record), random access (disks), and magnetic tapes.

- For sequential devices, simple sequential I/O is performed on the mounted TENEX file; for bidirectional (terminal-like) devices, two independent files (or a real terminal) are used. The mounted files may be declared as containing ASCII data, in which case the server translates the file's characters to or from the device's character set, or containing binary data, in which case no data transformation is performed. Data may be transferred a byte at a time (BIN and BOUT) or a record at a time (SIN and SOUT). A terminal-like device may be declared half-duplex, in which case the server echoes all input to the output (file) as it is read.

- For random-access (disk-like) devices, mounted files are assumed to be binary with sequential, fixed-length records. The relevant operations are BIN, BOUT, SIN, SOUT, SFPTR, and RFPTR.

- For "magnetic-tape" devices, either a real magnetic tape unit or a disk file may be mounted. A magnetic-tape disk file is read or written with both data and structure information intermixed in a private format that requires a byte size of 9. Tape operations are limited to DUMPI, DUMPO, and MTOPR.

In all cases where different forms of files are allowed (ASCII or binary for sequential devices, magnetic tape unit or disk file for tape-like devices), the I/O server handles the difference transparent to the emulator.

2.4.2 I/O Calls

The first word of a call block is used to pass an operation code from the emulator to the I/O server (in the right half) and to return completion status from the server to the emulator (in the left half); the emulator clears the status bits before issuing the service call and tests them on its completion. The remaining three words of a call block contain parameters and replies specific to each operation. The reader familiar with the TENEX system may notice a strong resemblance between the call codes and JSYS numbers and between the call parameter words (P1, P2, and P3) and JSYS accumulators (AC1, AC2, and AC3). The format of the first word of a call block is:

B0-B6	(not used)
B7	write-protected (input-only file)
B8	at end of tape
B9	at load point
B10	at file mark
B11	record-length error
B10-B11	00 record matches buffer size
	01 record less than buffer size
	10 file mark encountered instead of record
	11 record exceeds buffer size
B12	(not used)
B13-B14	(valid only as GTSTS replies)
B15	TENEX end-of-file
B16	call aborted
B17	call completed
B18-B35	call code (see below)

Bits B7 through B11 apply only to magnetic tape operations; B15 through B17 apply to all operations.

All operations that refer to a particular device take a PDN in parameter P1. The PDN is an identifying handle similar to a TENEX JFN.

Operations that transfer data to or from a buffer in buffer memory take a PDP-10 byte pointer as a parameter; the pointer addresses the byte *before* the first byte of the buffer (in anticipation of a PDP-10 ILDB or IDPB instruction). A byte pointer whose left half is zero causes one byte to be transfered per PDP-10 word (starting at the indicated word, not the next one); a byte pointer whose left half is all one-bits causes transfers to follow the standard ASCII text packing for the PDP-10.

The I/O calls are listed below; parameters returned by the I/O server are enclosed in parentheses; parameters reset by the I/O server are enclosed in braces:

Code	Name	P1	P2	P3
14	RSTAT	target PC	clock	call block address
22	CLOSE	PDN	---	---
24	GTSTS	PDN	---	---
27	SFPTR	PDN	record number	record size
43	RFPTR	PDN	(record number)	record size
50	BIN	PDN	(byte)	---
51	BOUT	PDN	byte	---
52	SIN	PDN	byte pointer	record size
53	SOUT	PDN	byte pointer	record size
65	DUMPI	PDN	byte pointer	{record size}
66	DUMPO	PDN	byte pointer	record size
77	MTOPR	PDN	operation	count
147	RESET	---	---	---

2.4.2.1 RSTAT

RSTAT provides the framework with information about the state of the emulated machine. The target PC (in P1) and high-resolution virtual timer (in P2) are always included in the call. If the emulator is currently waiting for a call to complete, P3 has the address of that call block; if it is not waiting, or cannot determine its state, P3 has zero.

RSTAT has two distinct uses: for responding to a STATUS action request (see Section 2.2.1 and Section 2.4.3.1) and (if necessary) for sychronizing emulated (virtual) time with the PRIM framework. Synchronization requires that at the end of each scheduled RSTAT interval the emulator wait for the previous synchronizing RSTAT call to complete and then issue a new synchronizing RSTAT call. The reporting RSTAT call and synchronizing RSTAT call should use different call blocks so that a status report can be made while awaiting completion of the previous synchronizing call.

2.4.2.2 CLOSE (all devices)

CLOSE closes the TENEX file(s) associated with the designated PRIM device, leaving the device with no files mounted.

2.4.2.3 GTSTS (all devices)

GTSTS returns status bits in the left half of the first word of the call block; two status bits are specific to this call:

B13	off-line (no files mounted)
B14	input waiting to be read

2.4.2.4 SFPTR and RFPTR (primarily for disk-type devices)

SFPTR positions the mounted TENEX file to the beginning of the record specified by P2 (*i.e.*, to the TENEX position P2*P3); if P2 is all one-bits, the file is positioned at its end and the number of records in the file is returned in P2. RFPTR returns the current record number (TENEX position/P3). For both of these operations, a negative (or zero) P3 is taken to represent a one-byte record. If SFPTR references a sequential device (see Section 2.4.1), it is applied only to the input file.

2.4.2.5 BIN and BOUT (primarily for sequential devices)

BIN reads one character from the (input) file mounted on the device; an end-of-file status at completion indicates there had been no more characters to read. BOUT writes one character to the (output) file.

2.4.2.6 SIN and SOUT (sequential or disk-type devices)

SIN transfers one record (of P3 bytes) from the (input) file to the buffer. SOUT transfers one record (of P3 bytes) from the buffer to the (output) file. For SIN, an end-of-file status at completion indicates that no data was transferred, since the file had been positioned at its end.

Binary files are assumed to be pure data (no structure information). A short (last) input record is padded with zero bytes.

ASCII files are processed one text line at a time, regardless of line length. Each SIN reads through the next end-of-line (truncating lines longer than the buffer or padding shorter ones with spaces), then translates the line and stores it in the buffer as a fixed length record of the requested size; line terminators are not part of the translated lines. Each SOUT causes the buffer to be translated, written into the file (with trailing spaces possibly stripped), and followed by an end-of-line sequence (carriage return followed by line feed).

2.4.2.7 DUMPI and DUMPO (magnetic tape device only)

DUMPI reads one record to the buffer from a real magnetic tape or a disc file specially encoded to contain tape-structure information as well as data. The number of bytes transferred is the lesser of P3 and the actual record length; bits B10 and B11 of the status word together indicate which of these governed the transfer. When the record is shorter than the buffer (B10,B11 = 01), the actual record length is returned in P3; when the record is longer than the buffer (B10,B11 = 11), the number of lost frames is returned in P3. DUMPO writes one record (of P3 bytes) onto a real magnetic tape or a specially encoded disk file.

2.4.2.8 MTOPR (magnetic tape device only)

The following MTOPR operations are implemented for magnetic tape devices; they do not use P3:

1	rewind
3	write EOF
6	forward-space one record
7	back-space one record
13	write gap
16	forward-space one file
17	back-space one file

The following MTOPR operations are implemented for sequential devices; P3 contains a repeat count:

12 write P3 ASCII line feeds (with no CR)
14 write P3 ASCII form feeds (with no CR)
15 write P3 ASCII carriage returns
37 write one ASCII CR followed by P3 LF's

They are useful where the device's character set does not contain form-control characters. If the file mounted on the device is binary, these MTOPR operations are ignored.

2.4.2.9 RESET

RESET requests the I/O server to abort all outstanding service requests. When the RESET call is completed, all prior calls on the server are guaranteed complete. For each outstanding call completed prematurely by the RESET call, the call-aborted status bit will be set. RESET is the *only* call guaranteed to complete in a short time.

2.4.3 Call Completion

When the I/O server completes a call, it sets the status bits and reply words in the call block. Until the call-completed bit is set, the call is outstanding and the call block logically belongs to the server. In general, the I/O server supplies the emulator with an error-free I/O interface. If a file problem occurs (*e.g.*, file not mounted, untranslatable characters) the framework requests the emulator to stop via the QUIT action request and reports the error to the user. Unless the user aborts the operation with a CANCEL command, the outstanding request will be retried by the server when emulation is resumed. Emulation errors (*e.g.*, use of an unrecognized PDN) are treated similarly, except that the server automatically aborts the request.

2.4.3.1 Waiting for Completion

Since I/O service calls are processed asynchronously, a call can take an indeterminate amount of time to complete; some calls may never complete. For operations that have no fixed emulated completion time (*e.g.*, input of a character from an emulated operator's console), the device handler must poll the call block by rescheduling itself to test the call-completed status bit again after some reasonable interval. For the majority of service requests, however, the emulated time of completion is fixed since the operation takes a known interval. When the scheduled time for completion arrives, the device handler must wait for the call-completed bit in the call block, thus synchronizing the I/O server with the emulated time frame. It is ***IMPERATIVE*** that ***ALL*** such waits include the ability to respond to QUIT and STATUS action requests before the request is completed. The *only* allowable exception to the above is a wait for a RESET call, since it is guaranteed to complete shortly.

2.4.3.2 Aborted Requests

An I/O server request can be aborted by the user-level CANCEL command, by an emulator RESET request, or by the server itself. When any of these happens, the server sets both the call-completed and call-aborted bits, indicating that it is done with the call block but the call was terminated prematurely. Should the emulator wish to abort a specific outstanding service request (*e.g.*, because the emulated device was reset), it may make the server call:

```
R.37 ← call block address + 400000000000 ! sign bit is set
CALL MLP.CALL ;
```

The server will then abort that call shortly (unless it completes normally first).

2.5 Breakpointing

An emulator must constantly look for conditions that will cause a break at the end of an emulation cycle. These break conditions fall into two classes: references and events.

A reference break is caused by some emulated reference to a target location in which a reference-break meta-bit is set. The locations subject to reference breaks, and the type(s) of references to be monitored there, are indicated by the tool builder in the emulator's descriptor tables. Reference-break meta-bits are permitted in working memory or MLP-900 auxiliary memory. The reference types are write (any modification), read (data fetch), and execute. All three types of breaks should be allowed in target memory; other locations, such as target registers, may be limited as deemed reasonable. A reference break does not suppress or interrupt the reference; rather, the execution cycle is completed normally but the occurrence of the reference break is logged for the debugger to process when the emulator stops at the end of that cycle. (Interrupting execution at the end of the cycle is consistent with the requirement for stopping cleanly and also avoids a problem that would arise if execution were interrupted in mid-cycle on the occurrence of the break condition--that of responding to possible changes made in the context during the break while not reporting the same breakpoint repeatedly.) The meta-bits are the four high-order bits of the 36-bit word:

B0: write break.
B1: read break.
B2: execute break.
B3: not used.

An event break is caused by the occurrence of any of a set of predefined events for which a corresponding meta-bit is set. One meta-bit is assigned in the context for each type of event. These meta-bits are described in the emulator's descriptor tables. An event break does not suppress or interrupt the event; rather, the execution cycle is completed normally but the occurrence of the event break is logged for the debugger to process when the emulator stops at the end of that cycle. The list of events is to include the following, and anything else deemed reasonable:

- Anomaly: any occurrence of a predefined program anomaly.
- Store: any memory store.
- Jump: any (successful) jump/branch.
- Step: any CPU instruction execution (*i.e.*, CPU single step).
- I/O: any I/O channel activity (*i.e.*, I/O single step).
- Interrupt: any interrupt sequence.
- Tick: every tick of the emulated clock(s) or other reasonable interval (such as a millisecond) if there is no clock.

Some anomaly conditions may be forced automatically if they are considered of sufficient importance by the tool builder, thus needing no associated meta-bits.

The occurrence of a break of either type is recorded in one word of an eight word (or larger) break-buffer. Each emulator cycle may use the entire buffer, from the beginning; the buffer is cleared by the debugger before resuming after a breakpoint. The format of a break-buffer word is

Break Type	B0 - B2	B3 - B8	B9 - B35
(Unused entry)	0	0	0
Event	0	Event index	Event parameter
Write break	1	Space number	Target address
Read break	2	Space number	Target address
Execute break	3	Space number	Target address

The tool builder assigns event indices to message strings that are contained in the emulator's descriptor tables. Event parameters, if any, are specific to each event. The space number corresponds to the SPACE declaration on which the referenced target address is declared in the emulator's descriptor tables (see Section 2.7).

2.6 Emulator Control Structure

The top-level structure of an emulator is assumed to be

```
initialize emulator ;
FOREVER DO
BEGIN
  IF reason to stop
  THEN BEGIN
    stop ;
    respond to switches and buttons ;
    END ;
  IF time to serve next scheduled device
  THEN service scheduled devices
  ELSE cycle mainframe ;
END ;
```

Except for the top-level loop shown in the control structure, an emulator may not have any loop that can run for an arbitrarily long amount of time. Any emulated operation that can take such long times must be prepared to abort if a STATUS or QUIT action request occurs. Each of the italicized phrases in the control structure is described below.

2.6.1 "Initialize Emulator"

Initialization involves the setting up of locations, such as mask registers, whose values are constant, although possibly a function of a configuration parameter. Inappropriate configuration parameters may be transformed in the process but may not be destroyed as the emulator must be re-initializable without harm.

The set of 64 device slots may be scanned to examine the configuration of installed devices and to initialize them properly (installation of additional devices is not permitted after emulator initialization). Pseudo-devices (*e.g.*, clocks) must also be initialized.

2.6.2 "Reason to Stop"

Emulation must be suspended when the target machine halts, when any break condition has occurred, or when the PRIM framework requests termination. When the emulator encounters a break condition during the emulation cycle, it logs the break for the debugger to process and flags a break state so as to stop at the end of that cycle. The framework requests termination (at the end of the next cycle) via action requests (see Section 3.3.3).

2.6.3 "Stop"

Before calling MLP.STOP, the emulator must leave the reason(s) for stopping in R.37, coded as follows:

Quit request	1
Emulated halt	2
Break(s)	4

If more than one stopping condition occurs in any cycle, the above numbers are OR'ed. Changes to the context made by the user during an emulator stop must appropriately affect the target machine on resumption of emulation. In particular, this requires that the emulator have no hidden copies of target state information when it stops.

2.6.4 "Respond to Switches and Buttons"

Upon resuming emulation after a stop, the emulator must check all manually settable switches that are not checked in the course of the normal emulation cycle (*e.g.*, master clear, load, *etc.*), and react appropriately. The emulator must also make sure that all hidden locations that reflect user-addressable values are set up again (*e.g.*, a global interrupt-check flop). In general, anything that is user-addressable, and that the emulator assumes is constant during execution, must be checked at this time.

2.6.5 "Time to Serve Next Scheduled Device"

The basic execution cycle consists of either asynchronous device service or a unit of mainframe processing. Device service is scheduled relative to the high-resolution virtual timer. A convenient scheduling mechanism is to maintain a linked list ordered by service time (earliest first) within an array of devices. Clock pseudo-devices, including the one used for synchronization (if implemented), are most easily treated as though they were scheduled devices of a unique type that is not installed.

Given a 36-bit, continuous, circular, virtual timer and events scheduled over a time span requiring no more than 35 bits of that timer, the correct test to compare the timer (here called R.TIME) with the scheduled time of an event (here called R.SCHED) is:

```
R.TIME - R.SCHED \1 ;
GOTO +1   !  must let the shift settle into SHE
IF NOT SHE THEN
  COMMENT scheduled time of the event has arrived ;
  ...
  END;
```

The same test can be used to compare scheduled times for ordering an event list.

2.6.6 "Service Scheduled Devices"

Each configured device has, in word 5 of its associated device control block (see Section 2.3.1), an assigned handler type that is used to select the appropriate handler for the device. The tool builder specifies the handler type for each device in the emulator's descriptor tables. When a device is installed, its handler type is entered into the control block. Serving a device

consists of removing that device from the scheduled device list and calling the proper handler. Each device then reschedules itself for further service according to its timing parameters and state.

A simple implementation of the wait-loop requirement of Section 2.4.3.1 involves turning a wait for completion into a rescheduling at the currently scheduled time. This puts the device back at the head of the event list, forcing the main loop to serve it again and again until completion of the request, while allowing any required stop to occur in the main control loop.

2.6.7 "Cycle Mainframe"

All mainframe activities take place in a single time frame and thereby consume internal (virtual) time. The selection of the appropriate mainframe activity for any emulation cycle is a function of the machine's internal design and priorities. A target interrupt, however, should be treated as a separate emulation cycle, thus permitting a break to occur between the target machine's acceptance of the interrupt condition and the execution of the next instruction. During mainframe execution, the emulator should maintain a jump history queue that records the last few (at least sixteen) successful jumps in terms of old and new program-counter values, the values being recorded circularly in two parallel spaces, typically in auxiliary memory (see Section 2.7.2 for a discussion of spaces). The location of the most recent pair of entries must be maintained in a pointer made known to the debugger via the emulator's descriptor tables.

2.7 Emulator Descriptor Tables

PRIM requires each of its emulation tools to have an associated loadable descriptor-table file containing a data base that identifies necessary elements of the emulation and defines the target architecture as it appears to the user. This file supplies assembly language conventions for the representations of numbers, operators, symbols, character sets, and instructions. It also defines the names, locations, and structures of addressable assemblages of cells of the tool (such assemblages are referred to as "spaces"), along with other characteristics of its architecture. This descriptor-table file is loaded automatically during PRIM initialization; the tool builder may also load such a file explicitly with the TABLES command.

One of the principal functions of the tables is the identification and naming of all cells of interest. Briefly, a cell is a set of contiguous bits contained within a single 36-bit word in either the TENEX target fork (the emulator's virtual main memory) or the MLP context (the emulator's control memory and MLP-900 registers). A cell is identified for the PRIM exec or debugger by an "extended PDP-10 byte pointer" in which the P (position) and S (size) fields have their standard meaning, while I, X, and E are combined into a 23-bit extended address. The extended byte pointers for all the MLP-900 registers (*R..37*, *M..17*, *MISC..17*, *CE..137*, *F..377*, *P..7*, *S..17*, and *A..1777*) are predefined in the tables using the GPM names. The macros that generate extended byte pointers take two arguments, named *byteptr* and *bitspec*. *Byteptr* is either one of the predefined extended byte pointers or an extended address that implies a full-word byte pointer (P=0 and S=36). The optional *bitspec*, if specified, defines a sub-byte within the named byte. The two acceptable forms of *bitspec* are <*a*-*b*> and <*a*,*b*> where *a* and *b* are integer constants. In the first form, bits *a* through *b* are indicated, where bits are number from 0 starting at the high-order bit in a byte. In the second form, a sub-byte is indicated as being *b* bits wide, positioned in the byte with *a* bits below (to the right of) its low-order bit. Addresses in the range 0 through 777777 (octal) are in the target memory; addresses 1000000 through 1017777 (octal) are in the context; all other addresses are invalid.

2.7.1 Structure of the Descriptor-table Source File

The relocatable descriptor-table file is actually the result of assembling a descriptor-table source file containing calls to MACRO-10 macros. The definitions of the macros used to build the tables are kept in the file <MLP>TABLES.MAC; when assembled, this file produces the file TABLES.UNV, which must be referenced at the beginning of the descriptor-table source file by using the MACRO-10 directive ***SEARCH TABLES*** (note that this facility is not completely supported prior to version 50 of MACRO-10). Implicit in these table-building macros is the assumption that the prevailing radix is decimal. If the tool builder wishes, he may change this by using the BASE macro (but should ***not*** use the RADIX directive). Since MACRO-10 assembles the source file to produce the relocatable file, the MACRO-10 conventions must be observed for the representation of numbers, character strings, and symbols. Except where explicitly contraindicated, the tool builder may freely use all of the features of the MACRO-10 assembler.

The second line of the source file must be a call on the EMULATOR macro. This macro defines the name of the machine, the width of several of the primary registers, the predominant type of arithmetic used, and the timing of the machine:

EMULATOR ***emname, pcwid, inswid, chrwid, arithwidth, arithaddr, buflow, bufhi, mincyc***

- ***emname*** is the name of the emulator.
- ***pcwid***, ***inswid***, and ***chrwid*** define the bit widths of the program counter, the basic instruction, and characters in the prevailing character set of the machine, respectively.
- ***arithwidth*** and ***arithaddr*** define the number of bits and type of arithmetic with which the PRIM debugger is to evaluate input. Currently A.2COM and A.ADDR are the only supported arithmetics; the former effects two's complement arithmetic while the latter interprets operands as unsigned magnitudes (other routines may be added to PRIM as the need arises).
- ***buflow*** and ***bufhi*** delimit the region in target memory that may be allocated by PRIM for I/O buffer space (by the routine DV.BUFF) such that ***buflow*** < ***bufhi*** and both are in the closed interval [400000, 776777], octal.
- ***mincyc*** specifies the number of emulated nanoseconds between successive ticks of the high-resolution virtual timer of the machine.

The EMULATOR declaration is followed by definitions, in arbitrary order, of the tool's spaces and symbols, character set(s), break tables, number and expression syntax, assembly formats and opcodes, events, emulated devices, and tool parameters. The last line of the source file must contain the MACRO-10 directive

END

In the macro descriptions that follow, each formal argument that ends with the suffix ***tag*** has its corresponding actual argument converted into an assembly-time symbol by prefixing it with a period; all such tags must therefore be unique in the first five characters. Such tags never conflict with the internal symbols used by the table macros, so all valid MACRO-10 symbols are allowed.

2.7.2 Spaces

A space is a two-dimensional array of cells that have been grouped together for convenience or necessity. Each column (or vertical slice) of the space consists of cells of uniform width; the concatenation of all the cells in a row of a space constitutes a "location" in that space. The user addresses locations, not cells, although when there is only one column in the space, as is very common, location and cell are identical. Typically, all the large spaces correspond to obvious entities in the target machine--like the target machine's main memory or registers--and a few miscellaneous spaces hold the rest of the visible locations. The debugger operations *next* and *prior* treat rows of a space as being circularly ordered, whether or not there is any inherent ordering of the locations.

Cells in a single location, as well as locations in a single space, should be non-overlapping. Different spaces may map the same bits in different ways. The first space defined *must* correspond to the target machine's primary memory; it is designated as space zero. Ordering of subsequent spaces is important to PRIM only in that a space number that the emulator reports in a reference breakpoint must correspond to the ordinal position (starting with zero) of that space's declaration in the file. The declaration of a space begins with the call:

SPACE *spacetag, access, population, width, distance*

- *spacetag* is the name of the space, unique in the first five characters; it is equated to the space number rather than an address in the tables since all internal references to a space use its number. Those spaces used to communicate with the debugger are identified through unique *spacetags* reserved for them.
- *access* is a sublist of the keywords READ, WRITE, READBREAK, WRITEBREAK, and EXECUTEBREAK or the keywords ALL or NONE, indicating the access and breakpoint capabilities associated with this space. The first two refer to the user's ability to access and modify locations in this space (if READ is not specified, the space is bypassed in symbol lookup); the next three indicate which, if any, of the reference breakpoint types are supported in this space. For a multislice space, the debugger *break* command sets meta-bits in the PDP-10 word that contains the first cell of a location; for PDP-10 words that are addressed by cells in more than one space, breakpoint capabilities must be established only for the single space in which the emulator actually reports such breaks. For a space with multiple locations within a single PDP-10 word, one set of meta-bits is associated with all the locations in the word: thus meta-bits set by the debugger for one of the locations apply to all and supercede those set earlier even for a different one of the locations that share the word; the emulator cannot identify which of these locations is associated with the break. For the tool builder (*i.e.*, for a *whiz*--see Appendix A), all the debugger access checks are bypassed: no-read spaces can be used to add all the tool builder's symbols and locations in the tables without their interfering with the tool user.
- *population* is the number of locations in this space; they are numbered from 0 through *population*-1.
- *width* is the width, in bits, of the locations in this space.
- *distance* is an optional parameter of the form

RANGE(*min, max*)

that affects debugger symbolic output of addresses in this space for locations

having no corresponding symbols. With the exception of memory, which may have no machine symbols, all spaces are expected to be completely covered by their symbols, making RANGE unnecessary. An address in a space is output by the debugger using the closest defined symbol, provided that that symbol is within the range [symbol+*min*, symbol+*max*] inclusive, where *min* and *max* are both signed integers.

Each SPACE macro call is followed by an arbitrary number of mapping function and symbol-declaration macro calls to complete the definition of the space. The macro ENDSPACE may be used after any SPACE macro to force its immediate definition. Since each successive SPACE call completes the previous one, ENDSPACE is usually required only at the end of the last space defined.

2.7.2.1 Symbols

A symbol is the name by which the user knows a location. Each such symbol input to the debugger is translated to a (*space*, *index*) pair, where *index* is an integer between 0 and *population*-1. Address arithmetic can then be performed on the *index* part. On output the debugger translates a (*space*, *index*) pair to a symbol or to a symbol with offset (see RANGE above).

Each SPACE declaration is followed by a list of SYMBOL macros declaring its associated symbols. A simple symbol declaration consists of just a *name* and *value* (index), representing a single symbol. Simple symbol entries are also created by the CELL, PROGRAMCOUNTER, and STEPFLOP macros, which are described later in this section. A more complex declaration can reference a recognizer function to designate a family of similar symbols, each of whose composite name consists of a leading substring equal to the family (SYMBOL macro) *name* and a trailing substring that is recognized or produced by the recognizer (which also is responsible for the construction or decomposition of the space index from/to the symbol value). Multiple symbols within a space may correspond to the same index; all are valid for input but only the first one declared is used for output. Symbols are entered into the currently open space using the SYMBOL macro:

SYMBOL *name*, *value*, <*rcfunc*(*arg1*, *arg2*, ..., *argN*)>, *distance*

- *name* is an arbitrary name used in other macros to reference this symbol.
- *value* is the row number (index) in the space where a simple symbol is located or is used by a recognizer function to generate an index.
- *distance* is an optional parameter of the form

RANGE(*min*, *max*)

that overrides the space's *distance* with respect to this symbol only. If unspecified, the *distance* of the space will be used.

There are currently seven recognizer functions implemented in PRIM (in the BLISS module SYMBOL); new functions can be added if needed. The BLISS routine names for the recognizer functions are of the form RC.xxx; the corresponding recognizer macros, with names of the form RCxxx, are described below.

Recognizer call	Function
RCOCT(*min, max*)	Parses an octal-number string, *n*, between *min* and *max* inclusive. Returns an index of: *value* + *n*.
RCDEC(*min, max*)	Similar to RCOCT, but with a decimal-number string.
RCHEX(*min, max*)	Similar to RCOCT, but hexadecimal.
RCNWRD(*min, max, b, d, r*)	Parses a base-*b* number, *n*, between *min* and *max* provided that *n* (mod *d*) = *r*. Returns an index of: *value* + (*n* / *d*).
RCMUL(*b, i, m*)	Parses a base-*b* number, *n*, between 0 and *m*, inclusive. Return an index of: *value* + (*n* * *i*).
RCSTR(<*string*>)	Parses no further input; instead, evaluates the ASCII *string* and returns its value as the index. This is an input recognizer only. It permits a symbol to be defined in terms of an expression involving other symbols.
RCOPN( )	Parses no further input; instead, returns the index of the open location, provided that it is in this space. This is an input recognizer only.

To use recognizer functions that do not have predefined macros, use

RCEXT(RC.*func, argument*)

where RC.*func* is the name of the corresponding BLISS routine and *argument* is a 36-bit value that may be the address of an argument vector.

2.7.2.2 Mapping Functions

Each slice (or column) of a space is specified using a mapping function that will translate an index for that space into an extended byte pointer to the corresponding cell. For spaces with more than one slice, the mapping functions are specified in order from the high-order byte of each location through the low-order byte. (The slices of a space may have different widths.)

Currently there are two mapping functions implemented (routines M.DEF and M.PTR in the BLISS module XVAL); new functions can be added if needed. M.DEF uses a set of five parameters to compute an extended byte pointer from an index, while M.PTR simply indexes into an array of extended byte pointers.

The macro MDEF is used to describe a regular slice that can be handled by the M.DEF routine:

MDEF *baseaddress, width, shift, density, increment*

- *baseaddress* is the extended address of the word containing the first cell of the slice.
- *width* is the width of the slice in bits.

- ***shift*** is the number of low-order bits in the word that are not in the last (rightmost) cell.
- ***density*** is the number of cells per word (packed every ***width*** bits from high-order to low-order end).
- ***increment*** is the value added to ***baseaddress*** to go from word to word.

All but the first argument may be omitted, with ***width*** defaulting to that of the space, ***shift*** defaulting to zero (indicating right-justification), ***density*** defaulting to one (indicating one cell per word), and ***increment*** defaulting to one (indicating that cells are in successive words). A symbol's space index I is translated by M.DEF into the P, S, and E fields of an extended byte pointer as follows:

Byte Location, P: $shift + (density - I \pmod{density} - 1) * width$
Width of Byte, S: $width$
Word Location, E: $baseaddress + increment * (I \,/\, density)$

The macro MPTR is used to describe a slice by a list of extended byte pointers, one per cell. MPTR is followed by ***population*** number of CELL macro calls (see SPACE and SYMBOL macros), each supplying one pointer; the cells are indexed in the order specified:

```
MPTR
CELL byteptr, bitspec, name
...
CELL byteptr, bitspec, name
ENDCELL
```

- ***byteptr*** and ***bitspec*** have been described previously.
- ***name*** is an optional argument; if supplied, it generates an implicit simple SYMBOL entry for this space using the given ***name*** and the index of this cell.

Newly implemented mapping functions may be referenced using the macro

MEXT ***function-name, argument-address***

- ***function-name*** is the name of the new mapping function.
- ***argument-address*** is the address of a block in memory containing its argument(s).

2.7.3 Distinguished Spaces, Locations, and Cells

Distinguished spaces are recognized through the use of one of the reserved ***spacetags***: OLDPCSPACE, NEWPCSPACE, BREAKBUFFER, and EVENTSPACE. The first three of these spaces need not have any defined symbols.

- OLDPCSPACE and NEWPCSPACE are a pair of spaces of equal size (preferably a power of two) with width equal to ***pcwid*** (see EMULATOR macro); they are used to hold the target machine's jump-history queue, with a parallel pair of locations holding the old and new program counter values for each jump. The debugger assumes they are used circularly in the forward direction (0, 1, . . . , ***population***-1, 0, . . .).
- TOPOFJUMPQ addresses the most recent entry in the circular buffers.
- BREAKBUFFER is a 36-bit space containing the encoded breakpoint descriptors reported to the debugger by the emulator.

- EVENTSPACE is a 1-bit-wide space that contains a location for each breakpoint event supported by the emulator; the debugger recognizes the events by their symbol names.

The following macros, which must each be used just once, inform PRIM of various distinguished locations that it uses implicitly. In addition to noting the distinguished locations for PRIM, they function as simple SYMBOL macros. The locations are, of course, also directly addressable:

PROGRAMCOUNTER *name, value*
STEPFLOP *name, value*

The PROGRAMCOUNTER macro specifies the location used in conjunction with the debugger ***go*** command and the information message produced when the emulator stops. The STEPFLOP macro specifies the location of the single-step event flag for use by the debugger ***single-step*** command.

The following macros, which must each be used just once, also inform PRIM of various distinguished cells that it uses implicitly. The cells have no user-known names, although the same bits can appear in some other space also:

CLOCK *byteptr, bitspec*
TOPOFJUMPQ *byteptr, bitspec*

The CLOCK macro declares the emulator clock that keeps virtual time; its unit is ***mincyc*** (see EMULATOR macro); its value is used to keep track of target time. The TOPOFJUMPQ macro declares a pointer to the top of the jump-history queue; its contents are used by the debugger ***jump-history*** command to identify the locations within OLDPCSPACE and NEWPCSPACE describing the most recent jump taken by the target machine.

2.7.4 Events

When the emulator detects and reports an event break, the debugger uses the event table to decipher the 6-bit event code and respond to the event. Each call to the EVENT macro generates one entry in the event table; no ordering is assumed. This event table defines the correspondence between event codes and event control bits (which are the locations in EVENTSPACE):

EVENT *code, prefix, parmtype, suffix, evaddr, spacetype*

- ***code*** is the event code reported to the debugger by the emulator.
- ***prefix*** and ***suffix*** are quoted strings to be output to the user by the debugger.
- ***parmtype*** interprets the event parameter accompanying the event; it is one of NONE, NUMBER, or a ***spacetag*** (implying the address of a location in that space).
- ***evaddr*** is either empty or the index into EVENTSPACE for this event's control bit, which is used by the debugger to check whether a breakpoint was set for this event and whether a break program is associated with it. Events that are not selectable by the user have no associated control bit in EVENTSPACE (and, therefore, can have no break program).
- ***spacetype*** is one of the following: NONE (or empty), NUMBER, or INSTRUCTION.

When an event break occurs, the debugger produces a message (based on the event *code*) consisting of the corresponding *prefix* string followed by the parameter value, output according to *parmtype*, followed by the contents of the location pointed to by the parameter, output according to *spacetype* (provided that *parmtype* is a *spacetag*), followed by the *suffix* string.

2.7.5 Character Sets

Several character sets may be employed within each target machine. The CHARACTERSET macro describes the several available character sets to the PRIM framework. In text mode the debugger uses the first character set defined, assuming that characters of width *chrwid* (see EMULATOR macro) are packed in locations. Any character set *chartag* may be referenced by RADIX and DEVICE macros. Each non-ASCII character set is defined using the following sequence:

CHARACTERSET *chartag*
CHARS 0,<*characters in ascending ordinal value*>
CHARS *m*,<*more characters, from the mth char*>
. . .
CHARS *n*,<*last characters in the character set*>
ENDCHARACTERSET

For an ASCII character set, the following declaration is used:

CHARACTERSET ASCII

The character set name, *chartag*, must be a valid MACRO-10 symbol unique in the first five characters. The resulting translation table for a character set is organized such that the ASCII character corresponding to the *i*th character in the character set is entered as the *i*th character in the table. Hence, the emulator builder supplies a list of ASC characters in the order of increasing ordinal value of the corresponding characters in the given character set. For ASCII, a character set entry is built but the table is not.

The first argument to a CHARS macro indicates the ordinal value of the next character specified. If this value is greater than the number of characters generated thus far for the character set, an appropriate number of padding characters will be inserted first. The padding character is defaulted for each character set as an output-only ASCII blank; it may be changed at any time using the macro

FILLER <*character*>

In this manner, sparse character sets may be specified compactly.

Due to MACRO-10 macro constraints, the following conventions have been adopted to specify certain characters in a character set. Control characters follow the BLISS-10 convention, using the question mark notation. Thus ?? is a question mark, ?C is *control-C*, ?G is *control-G*, ?0 is *null*, ?1 is *rubout*, *etc.*; additionally, ?(and ?) are used to represent < and >, which would otherwise interfere with the MACRO-10 scanner.

Some target-machine characters might not have ASCII equivalents. In such cases some ASCII character must be supplied to facilitate translation of output. An apostrophe immediately preceding a CHARS character declares that character to be for output only. That target

character will translate into the designated ASCII equivalent but that ASCII character will not translate back into the original character. For example, '?E occurring in position 12 in the character set will allow the target character whose value is 12 to be translated to ***control-E*** when expressed in ASCII, but translation in the reverse direction will be prohibited. To enter a literal apostrophe, precede it with a question mark, ***i.e.,*** ?'.

ENDCHARACTERSET is an optional macro that need only be supplied to cause immediate definition of the character set. A good practice is to place one after the last character set definition.

2.7.6 Break Tables

To aid the PRIM debugger in parsing expressions input by the user, five bit-encoded character-break tables are used: STARTSYMBOL, INSYMBOL, STARTNUMBER, STARTOPERATOR, and INOPCODE. All of these have identical calling sequences. STARTSYMBOL should contain all characters that can start a target-system symbol. INSYMBOL should contain all characters that can follow the first caracter of such a symbol. STARTNUMBER should contain all characters that can start a target-system number (not counting any prefix string that might be specified). STARTOPERATOR should contain all characters that can start a target-system operator. And INOPCODE should contain all characters that can appear anywhere in a target-system opcode. An example is:

INOPCODE <ABDX>

which declares that characters A, B, D, and X are the only characters occurring in any opcode. The question-mark convention may be used to enter control characters as well as special ones, though their occurrence in legitimate input atoms is improbable.

2.7.7 Numbers

The RADIX macro is used to describe to the PRIM debugger the target-system assembly-language syntax for both numbers and character constants, in both cases assumed to be a fixed ***prefixstring*** followed by digits or characters followed by a fixed ***suffixstring***, where either (or both) of the strings may be empty. For numeric constants, a ***base*** up to 36 is allowed, using the set {0, 1, . . . , 9, A, . . . , Z} for the digits, in the call:

RADIX ***prefixstring, base, suffixstring***

A character constant normally would use a ***base*** equal to the character-set size:

RADIX ***prefixstring, base, suffixstring, chartag***

The base of input numbers is self-defining as they must satisfy the syntax contained in the table of radices that drives the parsing; output numbers are generated according to the RADIX specification for those numeric radices supplied or as pure digit strings for other radices. (There is no character-constant output.)

2.7.8 Expression Evaluation

The operators in the target machine's assembly language are defined, along with their precedence, using the macros UNARY and BINARY:

UNARY *functionname, string, precedence*
BINARY *functionname, string, precedence*

- ***string*** contains the arithmetic operator in the target assembly language, or an invented name for use in PRIM, enclosed by delimiting characters.
- ***functionname*** is the name of an arithmetic function supported by PRIM that corresponds to the target operation being declared (see list of supported functions, below); all unary operators are prefix and all binary operators are infix.
- ***precedence*** reflects the relative binding strength of the declared operator; larger values take precedence over smaller ones.

The following functions are currently implemented:

Function Name	Arguments	Description
OP.ADD	2	Addition
OP.SUB	2	Subtraction
OP.MUL	2	Multiplication
OP.DIV	2	Division
OP.MOD	2	Modulus (remainder)
OP.LSS	2	Less than
OP.LEQ	2	Less than or equal to
OP.EQL	2	Equal to
OP.NEQ	2	Not equal to
OP.GEQ	2	Greater than or equal to
OP.GTR	2	Greater than
OP.AND	2	Bitwise AND
OP.OR	2	Bitwise OR
OP.NOT	2	Bitwise NOT
OP.XOR	2	Bitwise Exclusive OR
OP.CON	1	Contents of
OP.ABS	1	Absolute value
OP.NEG	1	Negation

Parentheses native to the target machine's assembly language or "invented" for use in PRIM may be declared by

PARENS *openingparen, closingparen*

where the *openingparen* and *closingparen* must each be enclosed by delimiter characters. An example of the use of the PARENS macro is

PARENS "(", ")"

2.7.9 Machine Instructions

The debugger's instruction assembler/disassembler is driven by a table of machine instruction formats. These formats use a set of parsing rules and a set of instruction-field descriptors. Fields, rules, and formats are each separately described below. Fields and rules can be defined in arbitrary order and are referenced by tags in formats and rules; formats must be collected into a single table. Symbolic opcodes are associated with instruction formats in a manner similar to the association of machine symbols with spaces.

2.7.9.1 Instruction Fields

An instruction is treated as a contiguous sequence of bits of a length that is some (initially unknown) multiple of *inswid* (see EMULATOR macro). For purposes of field definition, the instruction bits are numbered consecutively from zero at the high-order bit; location boundaries and/or PDP-10 word boundaries are ignored. A field identifies a set of (not necessarily contiguous) bits that is being treated as a unit in some instruction; within a field there may be one or more subfields consisting of contiguous bits. Fields are declared as

FIELD *fieldtag, bitspec, function*

for a simple field, or as

FIELD *fieldtag, , function*
SUBFIELD *bitspec*
...
SUBFIELD *bitspec*
ENDFIELD

for a broken field.

- *fieldtag* must be a valid MACRO-10 symbol unique in the first five characters.
- *bitspec* is a bit specification for the field or subfield of the form <*a*-*b*>, where *a* is the high-order bit of the field and *b* is the low-order bit.
- *function*, if specified, is the name of an arithmetic conversion routine that converts numbers to bits and bits to numbers; if not specified, it defaults to the machine arithmetic routine, *arithaddr*, from the EMULATOR declaration.

Where a field consists of a list of subfields, the field itself is the concatenation of all the subfield bits, with the first subfield at the high-order end.

2.7.9.2 Parsing Rules

A rule is an ordered list of parsing primitives that operate for both input and output, specifying the contents of instruction fields on assembly and generating an instruction string on disassembly. The execution of a rule succeeds when each of its primitives in turn succeeds; when any one fails, the rule fails. Rules deal with sequences of symbolic-expression fields and delimiters, with allowance for alternative and optional fields. The RULE macro begins a rule; it is followed by the rule's primitives in order:

RULE *ruletag*
ruleprimitive1
ruleprimitive2
...
ENDRULE

There are six *ruleprimitives*: MARK, EXPRESSION, CALL, TRY, IS, and ISNOT. Each takes its own particular arguments and has its own criteria for succeeding or failing:

MARK <"*char*">

Assembly: parses a single character of input and succeeds if, and only if, that character matches *char*.
Disassembly: appends the argument *char* to the string; always succeeds. (Used for indicating assembly language delimiters.)

EXPRESSION *fieldtag*	Assembly: parses and evaluates an expression and stores its value into the field named by *fieldtab*; always succeeds (empty expressions are permitted). Disassembly: appends the value contained in the field named by *fieldtag* as an appropriate string; always succeeds.
CALL *rule1, rule2* TRY *rule1, rule2*	Assembly or Disassembly: calls *rule1* (as a subroutine) and, if it fails, calls *rule2*. CALL succeeds if, and only if, either of the called rules succeeds; TRY always succeeds. *Rule2* is optional; if absent it always fails.
IS *fieldtag, value*	Assembly: stores *value* in the field named by *fieldtag*; always succeeds. Disassembly: succeeds if, and only if, the field named by *fieldtag* contains *value*; no output.
ISNOT *fieldtag, value*	Assembly: does nothing; always succeeds. Disassembly: succeeds if, and only if, the field named by *fieldtag* does not contain *value*.

Note that the only primitive that directly causes rule failure on assembly is MARK, while IS and ISNOT are the only direct causes of failure on disassembly. When a called rule (one referenced by a CALL or TRY macro embedded in some rule, rather than one invoked directly via a FORMAT macro) fails, all of its side effects are undone--as are those of any rules it might have called. On assembly this includes values stored in fields by EXPRESSION or IS as well as all input characters parsed; on disassembly this includes characters added to the output string by MARK or EXPRESSION.

The width of an instruction, on input or output, is derived from the rightmost field that is referenced successfully.

As an example, the following two rules handle an optionally indexed address field, where the index is designated by a comma followed by an index-register specification and is zero if not present. The fields ADDRFIELD and INDXFIELD designate the address and index fields, respectively.

```
RULE INDXADDR
  EXPRESSION ADDRFIELD
  CALL INDXRULE, INDXPAD
ENDRULE

RULE INDXRULE
  MARK <",">
  ISNOT INDXFIELD,0
  EXPRESSION INDXFIELD
ENDRULE

RULE INDXPAD
  IS INDXFIELD,0
ENDRULE
```

The first rule always succeeds on input, since INDXPAD always does; the second rule accepts an index specification if a comma is present and allows an index expression to be output if the field is not zero. (Actually, INDXPAD could be omitted and the call changed to TRY INDXRULE, since the instruction string is initially cleared on input, provided that the index field is not needed to establish the length of the instruction.)

2.7.9.3 Formats and Opcodes

A format consists of an opcode field, a rule for parsing the rest of the instruction beyond the opcode, and a list of opcodes that can be found in the opcode field:

FORMAT *ruletag, fieldtag*
OPC *name, value, <rcfunc(argument-list)>*
...
OPC *name, value, <rcfunc(argument-list)>*
ENDFORMAT

where *ruletag* and *fieldtag* are tags (unique in the first five characters) of the primary parsing rule and opcode-field definition for this format. An OPC macro has arguments identical to a SYMBOL macro, but here *value* is the numeric operation code rather than an index into a space. On assembly, a format is selected when one of its opcode names is recognized; the opcode *value* is stored into the field named by *fieldtag* and the rule *ruletag* is called. On disassembly, a format is selected when the contents of the field named by *fieldtag* matches the *value* of one of the opcodes; the rule *ruletag* is then called to complete the output.

2.7.10 Devices

The descriptor table supplies the PRIM exec with device information required to install and mount emulated devices. Each device macro specifies the name and assorted characteristics of one device type:

DEVICE *name, type, ftype, bytesize, chartag, paramtag, min, max*

- *name* is a quoted string giving the name of the particular device.
- *type* is a 16-bit emulator handler type (see Section 2.3.2).
- *ftype* is one of the keywords: INPUT, OUTPUT, SINGLEIO, or TTY, indicating the number of file(s) that may be mounted on the device by PRIM and the direction of data flow.
- *bytesize* should be coded as zero (to indicate that only ASCII files are allowed), a reasonable byte size (less than 64, giving the default byte size for any binary file that is mounted), or 64 plus a reasonable byte size (to indicate a fixed byte size for any binary file that is mounted).
- *chartag* is the tag from the CHARACTERSET macro (indicating the natural character set of this device) or is empty (to indicate that there is no such set). When a character set is provided, ASCII text files may be mounted, with character translation performed by the PRIM framework; when there is no such set, only binary files are allowed. (If neither binary nor ASCII is allowed, we are in trouble.) The user is asked at MOUNT time for file characteristics only when the device entry leaves him any choices.
- *paramtag*, if not blank, is the tag of a parameter table used to complete installation of this device (parameters are described in the next section).

- ***min*** and ***max*** delimit the number of units that may/must be installed. When the two are unequal, the user is asked how many he wants; they default to 0,1.

The DEVCLASS macro may be used to associate a device controller with a group of installable devices. The call is identical to that of the DEVICE macro, except that ***min*** and ***max*** are not specified; it must be followed by a DEVICE declaration for each device in the group and terminated by an ENDDEVCLASS macro:

DEVCLASS ***devstr, typo, ftypo, bytesizo, charsot, paramtag***
DEVICE ***devstr, typo, ftypo, bytesizo, charsot, paramtag, min, max***
. . .
DEVICE ***devstr, typo, ftypo, bytesizo, charsot, paramtag, min, max***
ENDDEVCLASS

A zero (or empty) ***typo*** is taken to indicate a dummy controller that is not actually installed; its ***ftypo, bytesizo,*** and ***charsot*** are ignored. The parameters gathered for this device class at installation time are given to each of the actual devices in the class--whether or not it is treated as a dummy controller.

2.7.11 Tool Parameters

A list of parameters is associated with each installable device (device parameters) and with the target machine (global parameters).

PARAMS ***paramtag***
PARAM ***namo, string, ptypo, colltag, argtypo, argtag, dofval***
. . .
PARAM ***namo, string, ptypo, colltag, argtypo, argtag, dofval***
ENDPARAMS

- ***paramtag*** is the name, unique in the first five characters, used to reference the entire list of parameters; the global parameters are recognized by the reserved tag MACHINE.
- ***namo*** is a unique quoted name for each parameter in the list.
- ***string*** is an optional quoted (noise) string that describes the units of the parameter's value.
- ***ptypo*** is one of the keywords EXPLICIT, DEFAULT, or FIXED, defining the manner and timing of the setting of the parameter's value. FIXED and EXPLICIT parameters are gathered only at device installation time and, therefore, are not applicable to global parameters. EXPLICIT parameters are obtained from the user with no default allowed; they appear to be part of the INSTALL command itself. FIXED parameters are obtained from ***dofval*** without consulting the user (FIXED parameters need neither ***namo*** nor noise ***string***); they do not exist for the user. DEFAULT parameters are initialized at installation time to their default values; thereafter they may be altered by the user via the SET command and inspected via the SHOW command.
- ***colltag*** is the tag of the parameter cell.
- ***argtypo*** is one of the keywords IMMEDIATE, NUMERIC, or KEYWORD.
- ***argtag*** is the tag of a NUMERIC or KEYWORD macro (empty for IMMEDIATE).
- ***dofval*** is the parameter's default value.

An *argtype* of IMMEDIATE is used for a default parameter that is fully specified by its *name*; the *defval* is stored into *celltag*. It is convenient for simple switches, with two immediate parameters addressing the same parameter cell with opposite values.

An *argtype* of NUMERIC is used to convert between a user-supplied number and an internal value as directed by a NUMERIC macro with a *numtag* that matches *argtag*:

NUMERIC *numtag, multiplicand, divisor, offset, exponent*

The conversion from a user number to an internal value is:

$$[\ multiplicand * user\text{-}number\ /\ divisor\ ]^{exponent} + offset$$

where *exponent* must be either 1 or -1; the computation is done using integer arithmetic, with multiplication being performed before division.

An *argtype* of KEYWORD allows the user to choose an entry from a menu of keywords defined in the tables by a KEYWORD macro with a *keytag* matching the *argtag*:

KEYWORD *keytag*
KW *keyword, value, bits*
...
KW *keyword, value, bits*
ENDKEYWORD

Each *keyword* is a quoted string that, when recognized, causes the associated *value* to be used for the parameter value. *Bits* is not currently used, but is intended eventually to supply 18 bits of information to the keyword routine. ENDKEYWORD is optional, forcing the immediate definition of the keyword list.

2.7.12 Parameter Cells

Parameter values are stored in cells or, for devices, in pseudo-cells not in the actual context. The CELLPTR macro is used to define each parameter cell, which is referenced by its *celltag*.

CELLPTR *celltag, byteptr, bitspec, cfunction*

- *byteptr* and *bitspec* (defined previously) are the true pointers to a cell in configuration memory or auxiliary memory for the global parameters. For device parameters, *byteptr* and *bitspec* define a byte within the device's context. The context for a device includes both the device block in PRIM and the configuration block in target memory; the following byte pointers are standard (but not predefined):

1,<31,1>	Half-duplex switch (for TTY type only). This parameter is handled by the I/O server rather than the emulator. When it is set, the server echoes input characters as the are read.
4,<0,36>	Device parameter word.
5,<0,16>	Device address, of which the 8 high-order bits are called the channel number and the 8 low-order bits are called the unit number.

6,<0,36>	Buffer word, which is of the form XWD(buffer-size, buffer-address). The *cfunction* DV.BUFF (see below) allocates buffers and is usually used with the buffer-word cell pointer.
7,<0,36>	Timing parameter word giving the device speed.

The cell pointers that are built may, of course, subdivide these bytes as needed; in particular, the two parts of the device address are usually specified separately.

- *cfunction*, if specified, names a routine that may further modify the value to be stored in the cell. The conversion functions, named DV.xxx, are found in the BLISS module DEVICE. The only function of general interest is DV.BUFF, which converts a buffer size (the input number) into a buffer word by allocating successive buffers from the region between *buflow* and *bufhi* (see the EMULATOR macro call).

2.8 Emulator Installation

Installing a new emulation tool in PRIM requires the creation of four files in the <PRIM> directory on the system interfaced to the MLP-900:

- *tool*.SAV is the executable program that the user runs to get the emulation tool; it is an extremely small program that gets the PRIM framework, leaving the *tool* name in a fixed location. This file is most easily created by taking an existing such file and replacing its name with this tool's name.

- *tool*.BIN is a binary file produced by the GPM compiler; it contains all control memory code, constants for masks and auxiliary memory, and the starting address of the emulator's initialization code.

- *tool*.DESCRIPTOR-TABLE is the relocatable output file produced by assembling the emulator's descriptor-table source file (see Section 2.7); *tool* is also the *emname* used in the descriptor file.

- *tool*.CONFIGURATION is a PRIM SAVE file that contains the default target-system configuration--the default values for all global parameters and any universal devices or debugger formats that are to be available to all users as initial conditions. It is created by running PRIM, loading the emulator's descriptor-table file, setting all the parameters (configuring to the extent necessary), and then executing the SAVE CONFIGURATION command.

For each emulation tool, all uses of the name *tool*, above, must be identical.

News regarding an emulation tool may be posted by sending a message to the file <PRIM>PRIM.NEWS, using the group name "*tool*:", as in

tool:*<PRIM>PRIM.NEWS

on the system interfaced to the MLP-900.

Chapter 3
MLP-900 Reference Manual

This chapter describes the MLP-900 briefly and discusses its instructions. Although the emulator writer ordinarily will not be concerned with the bit-level descriptions of the machine instructions, the detailed descriptions are given for reference. It is suggested that on first reading the hardware-level discussions be skipped or skimmed. The low-level syntax and semantics discussions are useful background for the next chapter on the GPM language.

The MLP-900 is a large, vertical-word, microprogrammable computer designed as a general-purpose emulation host on which each user can create his own target machine. It is a synchronous machine with a 300-nanosecond cycle time, 4096 words of control memory, and a large set of internal registers. A number of original features help make the MLP-900 an exceptionally powerful microprogramming tool; principal among these are a subroutine stack, a multi-level interrupt mechanism, a two-state protection facility, paging and memory protection hardware, and provision for user-specified language boards to provide hardware assistance for particular applications (no user language boards currently exist or are contemplated, however). It is characterized by two parallel computing engines, known as the operating engine (OE) and the control engine (CE). The OE is a 36-bit-wide arithmetic and data-transfer machine; it includes the hardware for the main memory and external interfaces and the bulk of the register space, including a 1K internal (auxiliary) memory. The CE is the instruction-sequencing and control unit; it includes the stack-handling, interrupt, and protection mechanisms.

MLP-900 instructions are known as "ministeps;" each engine has its own unique instruction set. Ministep execution proceeds sequentially, either singly or in OE-CE pairs. An MLP-900 ministep is contained in 32 instruction bits, occupying the low-order bits of the 36 accessible bits in a control-memory word; the four high-order bits are used only in conjunction with long immediate OE instruction, where the second word contains a 36-bit literal constant. The first four bits of each ministep constitute the opcode and the next four, the sub-op; in general, the opcode determines the format of the remaining fields of that ministep. The high-order bit of the opcode designates the engine: 0 for an OE ministep, 1 for a CE ministep. At the beginning of each cycle, the CE fetches a pair of ministeps from control memory--from the current address and its successor--and examines them: if the first is an OE ministep and the second is a CE ministep, then the pair is executed during this cycle; otherwise only the first ministep is executed (the other will be the first ministep of the next cycle, barring a branch). This parallelism serves to increase the effective machine speed and, with two exceptions, is transparent to the user: first, interengine data transfers require execution of an OE-CE pair; second, CE registers modified as a side effect of an OE ministep cannot be sensed by a paired CE ministep that executes in the same cycle. Since all changes to the state of the machine occur simultaneously at the end of the cycle, all computations and decisions are based upon the values present at the beginning of the cycle.

The MLP-900 hardware recognizes two distinct execution states: "user" mode and "microvisor" (microprogram supervisor) mode. User-mode microcode is subject to three restrictions: privileged ministeps may not be executed, privileged registers (in both the OE and CE) may not be modified, and a branch to a microvisor location other than a designated entry point is illegal. Violation of any restriction results in a (privileged) interrupt and suppression of the current cycle. These restrictions fully protect the external interface, the main-memory protection and paging facility, and the microvisor itself from the user microcode; additionally,

the microcode is restricted from modifying itself. Since this manual is intended for the emulator developer, who will be writing MLP user-mode programs, privileged facilities are not discussed in detail.

The MLP-900 main memory interface includes a memory-protection and paging scheme that, together with some microvisor code, provides the user with a 256K virtual address space. This scheme mimics the memory management provided by the TENEX pager on the PDP-10.

3.1 Primary Language Symbols

There is no assembler for the MLP-900. Instead, machine instructions may be written as special low-level statements to be processed by the GPM compiler. The low-level statements for each machine instruction are described in this chapter. To define these low-level statements completely, it is necessary to introduce the primary symbols of the GPM language in this chapter rather than in the chapter on the GPM language itself. The GPM syntax equations are given in this chapter and the next as modified BNF definitions. Each definition is preceded by a definition number within braces; each reference to that definition is immediately followed by its definition number within braces so as to facilitate cross references. Semantic comments, where necessary, are enclosed in doubled angle-brackets immediately following the relevant definition. All syntax equations before program{63} are in this chapter; the remainder are in Chapter 4. The few primitive constructs referenced in definitions are given in italics, as in *emptystring*. GPM statements are composed of five primary symbols or syntactic entities:

- Identifiers (see Section 3.1.1)
- Reserved identifiers (see Section 3.1.2)
- Octal numbers (see Section 3.1.3)
- Blanks (see Section 3.1.4)
- Nonalphanumeric characters (see Section 3.1.5)

3.1.1 Identifiers

An identifier is a string of words (alphanumeric strings) or numbers connected by periods. The first field must not be a number, and words must nnot begin with a digit (0 - 7). The last all-numeric field is referred to as the index; it is used extensively for reserved identifiers (*e.g.*, R.0 stands for the first general register and R.17 stands for the sixteenth general register).

Syntax:

{1} id ::=
reserved-identifier | . word{2} | word{2} | id{1} . subid{4}

{2} word ::=
alpha{3} | word{2} alpha{3} | word{2} digit{6}

{3} alpha ::=
8 | 9 | A | B | ... | Y | Z | a | b | ... | y | z

{4} subid ::=
word{2} | number{5}

{5} number ::=
digit{6} | number{5} digit{6}

{6} digit ::=
0 | 1 | . . . | 6 | 7

3.1.2 Reserved Identifiers

Reserved identifiers have the same syntax as identifiers in GPM but additionally include all nonalphanumeric symbols (the nontrivial reserved identifiers are listed in Appendix C). In this and the next chapter, all reserved identifiers are shown in upper-case; an arbitrary member of a set of indexed reserved identifiers (*i.e.*, an identifier with any of its permitted index values) will be denoted by an italicized, indexed name where the index is given as the double-dotted upper limit, as in the example below.

Example:
There are 32 general registers (R.0 - R.37). The symbol *R..37* represents any one of the set of registers, {R.0, R.1, . . . , R.36, R.37}.

Indexed reserved identifiers are assumed to have zero origin. Reserved identifiers cannot be used as branch destinations (see Section 4.5.3) or as a title (see Section 4.1).

3.1.3 Numbers

All numbers in GPM, including identifier index fields, are octal. Thus A.1973 would be interpreted as the two identifiers A.1 and 973. The symbols 8 and 9 are always treated as letters.

3.1.4 Blanks

All nonprinting characters (space, tab, linefeed, carriage return, and form feed) are converted by GPM to blanks. Blanks separate numbers and identifiers; otherwise they have no syntactic or semantic function. There is one additional "blank character," an arbitrary string starting and ending with a percent sign (%). This "blank character" is not the preferred method of introducing a comment, as will be treated in more detail in the discussion of the GPM listing format in Section 4.7.

3.1.5 Nonalphanumeric Characters

All nonalphanumeric characters are reserved symbols. Except for the period, they are all self-terminating and cannot appear as part of an identifier.

3.1.6 Examples of Primary Symbols

The string R.1 ABC*1248X 12A.B;C.3.4.X will be interpreted as:

R.1	Reserved identifier with index of 1
ABC	Identifier
*	Character
124	Number
8X	Identifier

12	Number
A.B	Identifier
;	Character
C.3.4.X	Identifier with index of 4

3.2 Operating Engine

The Operating Engine (OE) is a 36-bit data-transfer and -manipulation engine; it also contains the interfaces to both main memory and the PDP-10 I/O bus. The computational facility consists of a three-input (two operands and a mask) "primary adder" capable of various arithmetic and boolean functions, a "primary shifter," and an "extension shifter" used for single- or double-word shifts. Operands are taken from, and results stored into, the general registers (*R..37*); masks are taken from the mask registers (*M..17*). One byte of CE flops (CE.14) is devoted to functions associated with the adder and shifter(s). The memory and I/O bus interfaces consist of a number of special registers (grouped together within *MISC..37*), the main-memory address translator (*XLATOR..777*), and the memory-referencing ministep (CEDE).

Note that in all OE ministeps involving a large constant operand, the ministep takes two control memory words; while the hardware handles the decode automatically, the programmer must be aware of the fact that such a ministep always executes without a paired CE cycle. A large constant is one that cannot be expressed in six bits (*i.e.*, not in the range 0-77, octal).

3.2.1 Operating Engine Operands

Table 3.1
Operating Engine Address Space

Group	Extension	Register	Mnemonic	Description
0000	---	xxxxx	*R..37*	General Registers
0001	---	-xxxx	*M..17*	Mask Registers
0010	---	xxxxx	*MISC..37*	Miscellaneous Reg.[1]
01xx	xxx	xxxxx	*A..1777*	Auxiliary Memory
1000	---	-----	XBUS	CE Exchange Bus
1001 and 1010	xxx	xxxxx	*XLATOR..777*	Translator Memory[2]

The OE operands are contained in one sparse 12-bit address space. In addition to the mnemonics shown in Table 3.1, these operands may be addressed as *OE..7777*. The OE registers may be addressed directly, or indirectly through the CE pointer registers. As the pointer registers are only 8 bits wide, the OE register group is specified in the instruction. There are two types of indirect referencing available. Normal indirect (@) uses the pointer value to select both the extension and the register. Special indirect (*) is similar, except that the low-order bit is forced to 1.

---

1. MISC.20-MISC.37 are privileged.
2. Privileged.

Examples:

R.0 ↕ P.5
XLATOR.400 @ P.7

A-Operands. An OE A-operand represents a reference to a general register (*R..37*) either as an explicitly stated general register or as an indirect reference through a pointer register (*P..7*). The encoding is shown in Figure 3.1.

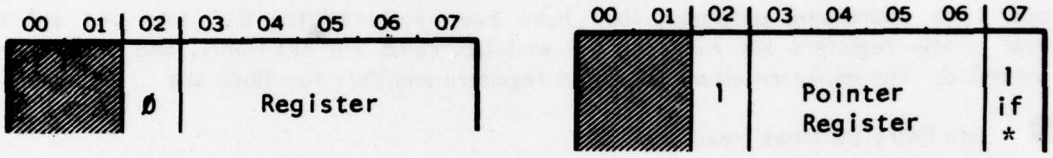

Figure 3.1 A-Operand Formats

Examples:

R.13
@ P.11
↕ P.7

B-Operands. An OE B-operand represents a reference to a general register (as in an A-operand), to a pointer register, or to an immediate operand. The encoding is shown in Figure 3.2.

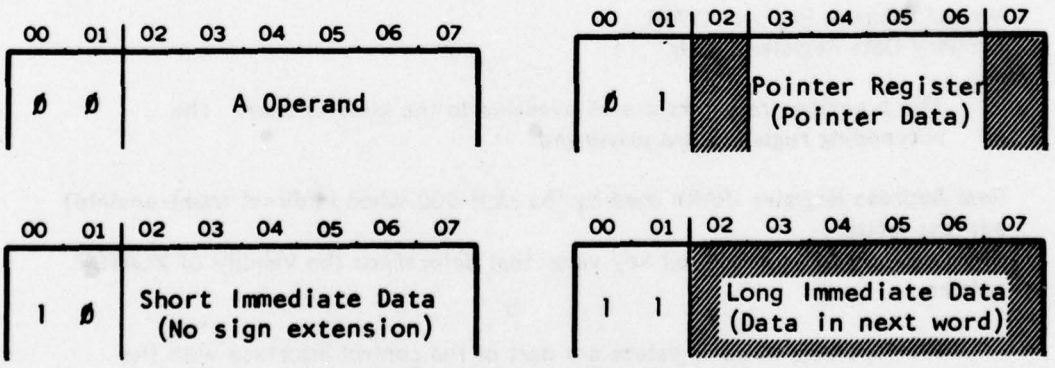

Figure 3.2 B-Operand Formats

3.2.1.1 *R..37* General Registers

There are 32 general registers (*R..37*), each 36 data bits wide. Four parity bits, one for each 9-bit byte, are maintained with each register. All 32 registers are addressable as inputs to the primary adder. Only R.37 (the Shift Extension Register) has a dedicated function.

3.2.1.2 *M..17* Mask Registers

There are 32 mask registers, but only *M..17* can be addressed by an OE instruction. The high-order bit of the mask address is taken from the protected CE flop MBS (F.167). User programs, therefore, see only 16 mask registers. The mask registers condition the adder functions to accomplish subword operations.

3.2.1.3 *MISC..37* Miscellaneous Registers

There are 32 miscellaneous registers (*MISC..37*) dedicated to a number of different functions. For addressing purposes, they have been gathered together into one set of registers. Some registers are readable and writable, some are read-only, and others are unimplemented. The implemented miscellaneous registers and their functions are

0 Data Entry Switches (read only)
1 Main Memory Address Switches (read only)
2 Processor Address Switches (read only)

The above three entries are pseudo-registers that make available the three sets of switches on the console. The following two registers can be read and written; they are tied into language boards.

4 Primary Instruction Register (PIR)
5 Secondary Instruction Register (SIR)

The folowing two registers are used in memory referencing. For more information, see the CEDE instruction (see Section 3.2.2.2).

16 Virtual Address Register (VAR)
17 Memory Data Register (MDR)

The preceding registers are all available to the general user. The succeeding registers are privileged.

23 Real Address Register (RAR): used by the MLP-900 when in direct (nontranslate) address mode.
31 Key Register: contains a 7-bit key value that determines the validity of XLATOR entries.

The following three registers are part of the control interface with the PDP-10 (see Section 3.4).

32 DATAO
33 DATAI
34 Command/Status Register

36 Virtual Address Compare Register (VADRC): compared to the virtual address (VAR) at every main memory reference, when enabled by SARM.1, and generates an action request (VADR, F.124) when a match occurs (see Section 3.3.3).

37 Control Memory Address Compare Register (CMADRC): compared to the memory address at every control memory reference, when enabled by SARM.0, and generates an action request (CMADR, F.110) when a match occurs (see Section 3.3.3).

A data transfer to an unimplemented register is a no-op; a data transfer from an unimplemented register yields -1.

3.2.1.4 *A..1777* or *A.PG..3* Auxiliary Memory

There are 1024 words of 60-nanosecond auxiliary memory, which can be used as a scratchpad or cache. In practice, auxiliary memory must be treated as consisting of four "pages" of 256 words each, since indirect references require the page to be specified in the instruction rather than in the pointer. *A.PG..3* are the origin words for the auxiliary memory pages (A.0, A.400, A.1000, and A.1400).

3.2.1.5 XBUS Exchange Bus

The OE exchange bus is a pseudo-register connected to the CE exchange bus (see Section 3.3.1.3.). Data transfers between the engines are accomplished by an OE-CE instruction pair, with the OE instruction either a GENT or a CEDE (which references the exchange bus), and the CE instruction either a MOVE (which references the exchange bus) or a BLOT (other than MOE). Since these instruction pairs are executed in parallel, the OE instruction (GENT or CEDE) must appear first regardless of the transfer direction. In transfers to the OE, any bits not loaded by the CE instruction are transferred as zero. In transfers to the CE, any bits not used by the CE instruction are ignored. A reference to the exchange bus without a paired CE instruction is undefined.

3.2.1.6 *XLATOR..777* Translator Memory

The translator memory consists of 512 20-bit words used to translate target-machine virtual addresses to real addresses in the PDP-10 memory. Each word consists of a 7-bit key value, an 11-bit real-page value, a write-permit bit, and a parity bit. Whenever translation is performed, the nine high-order bits of VAR are used as an index into the translator memory to select a translator word; this word is valid if the key value matches the key register (MISC.31) and if either the write-permit bit is on or this is a fetch. Note that a GENT from the translator memory reads the word selected according to the ***old*** value of VAR and ***then*** modifies the nine high-order bits of VAR to address the requested word, which is not read except by coincidence. Translator memory is privileged.

3.2.2 Operating Engine Operators

Four of the eight possible OE opcodes are defined. The other four produce undefined results, but the general flavor of their ministep decoding is the same. In particular, the B-operand decode (see Section 3.2.1) applies to ***all*** OE ministeps (even to defined ministeps that have no B-operand); whenever the B-operand specifies long immediate data, the following word is taken as a 36-bit literal rather than as a ministep. The OE operators are:

- GEAR (General Arithmetic). Performs binary arithmetic, logical operations, and single-register shifts.
- CEDE (Conditional External Data Exchange). Transfers addresses and data between the OE and main memory.
- SHIN (Shift Instruction). Performs various single- and double-register shifts, plus the iterated steps of multiply and divide loops.
- GENT (General Data Transfer). Transfers data between the OE registers and to and from the CE.

3.2.2.1 GEAR *GEneral ARithmetic*

This ministep provides arithmetic and logical capability involving the general registers, pointer registers, and constants. The GEAR internal coding is shown in Figure 3.3. The arithmetic codes are listed in Table 3.2. The shift-amount coding is found in Table 3.3. The test-mode and clear-mode bits are set to 1 when that mode is active. The A-operand (*aa* in Table 3.2) and the B-operand (*bb* in Table 3.2) are coded as described in Section 3.2.1.

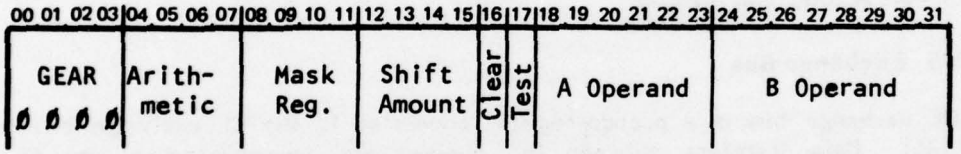

Figure 3.3 GEAR Ministep

Table 3.2
GEAR Arithmetic Codes

Code	Primary Adder Operation	
0	aa ← NOT aa OR bb	
1	aa ← NOT aa AND bb	
2	aa ← bb	
3	aa ← aa AND NOT bb	
4	aa ← aa OR NOT bb	
5	aa ← aa AND bb	
6	aa ← aa OR bb	
7	aa ← NOT bb	
10	aa ← aa XOR NOT bb	
11	aa ← aa + bb	
12	aa ← bb + ¬aa + 1	(bb - aa)
13	aa ← aa + bb + COF.1	(aa PLUS bb)
14	aa ← aa + ¬bb + COF.1	(aa MINUS bb)
15	aa ← bb + ¬aa + COF.1	(bb MINUS aa)
16	aa ← aa + ¬bb + 1	(aa - bb)
17	aa ← aa XOR bb	

The encoding for shift amounts for GEAR and SHIN ministeps is shown in Table 3.3.

Table 3.3
Shift Amount Encoding

Shift Amount (8)	Shift Amount (10)	Shift Code Left	Shift Code Right
0	0	10	0
1	1	11	1
2	2	12	2
4	4	13	3
6	6	14	4
10	8	15	5
14	12	16	6
20	16	17	7

Syntax:

```
{7}   gear ::=
          aa{8} ← gexp{9} gmod{11} ; | gexp{9} gmod{11} ;

{8}   aa ::=
          R..37 | @ P..7 | * P..7

{9}   gexp ::=
          aa + bb | aa - bb | bb - aa |
          aa PLUS bb | aa MINUS bb | bb MINUS aa |
          aa AND bb | NOT aa AND bb | aa AND NOT bb |
          aa OR bb | NOT aa OR bb | aa OR NOT bb |
          aa XOR bb | NOT aa XOR bb | NOT bb | bb
          <<see aa{8} and bb{10}>>
          <<when using the first form of gear{7}, aa here and there must be identical>>

{10}  bb ::=
          aa{8} | number{5} | P..7

{11}  gmod ::=
          amask{12} testmode{13} gshift{14} | gshift{14} amask{12} testmode{13} | ...
          <<amask, testmode, and gshift may be specified in any order>>

{12}  amask ::=
          ( M..17 ) | [ M..17 ] | emptystring

{13}  test ::=
          * | emptystring

{14}  gshift ::=
          shdir{15} samount{18} | emptystring

{15}  shdir ::=
          shleft{16} | shright{17}
```

{16} shleft ::=
LEFT | \

{17} shright ::=
RIGHT | /

{18} samount ::=
0 | 1 | 2 | 4 | 6 | 10 | 14 | 20

Examples:

```
R.1 ← R.1 + R.2;
R.7 ← R.7 - P.0 /1 [M.1] #;
R.37 ← 173 - R.37 \2 (M.2);
@P.0 ← @P.0 XOR NOT 3 (M.17);
#P.17 ← #P.17 AND P.3 /4 [M.27] #;
@P.3 ← NOT @P.3 OR R.17 \20 (M.21);
@P.1 ← #P.1 MINUS @P.1 (M.3) #;
```

Semantics:

The GEAR ministep is used for arithmetic operations. It selects two operands and a mask, routes them to the primary adder, and then specifies a shift of the result through the primary shifter. The result is then stored back into the A-operand (see Section 3.2.1 for a discussion of OE operands) in either clear or normal mode. This operation is shown in Figure 3.4.

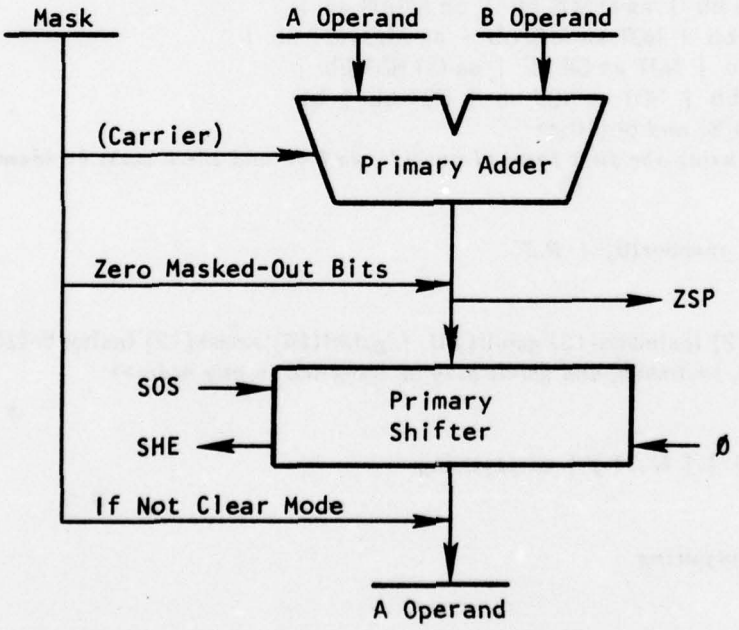

Figure 3.4 Operating Engine: GEAR

Mask

The requested operation is conditioned by the value of the specified mask register. A one-bit (1) in the mask is a masked-in bit; a zero-bit (0) in the mask is a masked-out bit. The default mask is M.0.

Adding under a mask. The primary adder treats all the masked-in bits as one contiguous operand field; carry generation is suppressed in masked-out bits, and carry propagates over masked-out bits. For all operations, the masked-out positions are forced to zero at the primary adder output. For the -, PLUS, or MINUS operators, the third term in Table 3.2 (either +1 or COF.1) is treated as a carry into the low-order (masked-in) bit.

Shifting under a mask. The shifter ignores the mask.

Storing under a mask. In Clear mode, [*M..17*], the entire 36-bit output of the primary shifter is stored; if the shift amount is zero, then all masked-out bits are necessarily cleared to zero. In normal mode, (*M..17*), only the masked-in bits are stored; the masked-out bits remain unchanged in the register.

Test Mode

If "aa ←" is not specified in the GEAR, or if the test mode modifier "*" is present, the store into the A-operand (see Section 3.2.1) is suppressed. In any case, all applicable flops (see Table 3.4) are set.

Operators

All valid operator combinations are listed in the syntax for *gexp* in Section 3.2.2.1. Normal addition (+) and subtraction (-) operators are two's complement; NOT is a logical operator (one's complement). PLUS and MINUS are one's complement operators and take flop COF.1 as an initial low-order carry-in; these operators can be used to produce multiple-precision results. Both the "-" and "MINUS" forms of subtraction are defined in terms of complementation, addition, and low-order carry-in; carry-out is always generated by addition.

Shifts

All valid shift amounts are listed in the syntax for *samount* in Section 3.2.2.1. The prefix "/" designates a right shift (divide) and the prefix "\" designates a left shift (multiply). The default shift is "RIGHT 0". The boundary shift conditions are shown in Figure 3.5.

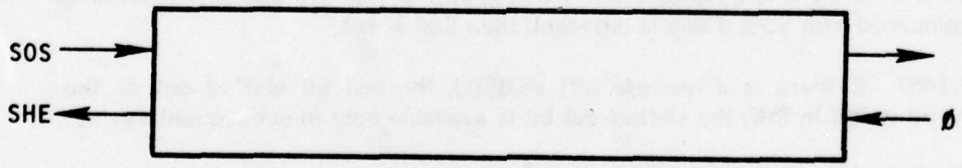

Figure 3.5 Shifter Boundary Conditions

Flip-Flops

Table 3.4 lists all flops involved in any GEAR.

Table 3.4
GEAR Flops

Flop	Active Condition
COP, COF.1, COF.2	+, -, PLUS, MINUS
ZSP, ZRF.1, ZRF.2	All GEAR operations
SOS	Nonzero shift
SOF, SHE	Nonzero left (\) shift

COP (F.300). This pseudo-flop contains the carry-out value for +, -, PLUS, and MINUS operations executed during the current cycle. It is valid only ***during*** the current cycle (*i.e.*, for testing by a paired CE instruction).

COF.1 (F.140). This flop contains the most recent setting of COP and thus has the carry-out value of the last +, -, PLUS, or MINUS operation completed.

COF.2 (F.141). This flop contains a copy of the previous setting of COF.1, and thus has the carry-out value of the next-to-last +, -, PLUS, or MINUS completed.

ZSP (F.301). This pseudo-flop is set if the masked output from the primary adder for this current operation is zero. Active for all GEAR operations, it is valid only ***during*** the current cycle.

ZRF.1 (F.142). This flop contains the most recent setting of ZSP (except in the case of PLUS and MINUS, when it is set to the logical product of ZSP and its own prior value) and thus reflects a zero result from the last GEAR completed.

ZRF.2 (F.143). This flop contains a copy of the previous setting of ZRF.1, thus reflecting a zero result from the next-to-last GEAR completed.

SOS (F.146). If there is a nonzero right shift (/), SOS is copied into the vacated bits.

SOF (F.147). If there is a nonzero left shift (\), each bit shifted out of the leftmost bit is compared with SOS; if any is different, then SOF is set.

SHE (F.145). If there is a nonzero left shift (\), the last bit shifted out of the leftmost bit is left in SHE; the shifted-out bit is available only in subsequent cycles.

3.2.2.2 CEDE Conditional External Data Exchange

CEDE is used to fetch and store main memory. All memory fetches or stores require the execution of two CEDEs. The first CEDE provides an address that is treated as virtual or real (depending on TRBY, F.165), initiates a translate cycle if virtual (*i.e.*, if not TRBY), and initiates the memory fetch if reading. The second CEDE, which need not follow immediately, provides the data for a store or waits for the data from a fetch. Page-fault action requests take place at the time of the second instruction (the wait or store) and cause that instruction to be suppressed.

The CEDE exchange code (see Table 3.5) determines the sub-op being executed. The A-operand and B-operand of FOP and SAD are identical to their coding in GEAR; the "Op A

Extend" and "Op A Group" fields are ignored. For WOP, SOP, and WOS, the A-operand specifies any OE register, the 12-bit address being coded in three sections (the 4-bit group, the 3-bit extension, and the 5-bit register); the operand may also be indirect through a pointer, in which case the indirect addressing is done within the indicated group and the "Op A Extend" is ignored. WOP, SOP, and WOS ignore the B-operand.

Test mode inhibits fetching, storing, translating, and the modification of any register, but waiting and page faulting are still performed. The subtract bit, when set, specifies two's complement subtraction instead of addition for those CEDEs that do arithmetic; the subtract bit is ignored for other CEDEs.

Table 3.5
CEDE Exchange Codes

2	FOP	Fetch Operand
3	SAD	Set Address
11	SOP	Store Operand
14	WOP	Wait for Operand
15	WOS	Wait for Operand, Stream Mode
16	ROW	Retry Operation

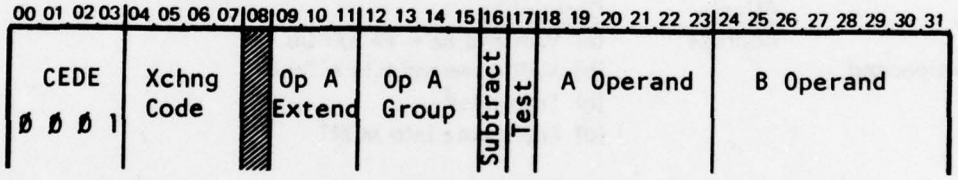

Figure 3.6 CEDE Ministep

Syntax:

```
{19}  cede ::=
          cedeA{20} | cedeB{23} | cedeC{28}

{20}  cedeA ::=
          cedeAcode{21} aa{8} sign{22} bb{10} testmode{13} ;

{21}  cedeAcode ::=
          FOP | SAD

{22}  sign ::=
          + | -

{23}  cedeB ::=
          cedeBcode{24} oeloc{25} testmode{13} ;

{24}  cedeBcode ::=
          SOP | WOP | WOS
```

{25} oeloc ::=
oereg{26} | oepage{27} @ *P..7* | oepage{27} * *P..7* | XBUS

{26} oereg ::=
R...37 | *MISC...37* | *M..17* | *Λ..1777* | *XLATOR..777*

{27} oepage ::=
oereg{26} | *Λ.PG..3* | *XLATOR.PG..1*

{28} cedeC ::=
ROW testmode{13} ;

Examples:

```
FOP R.3 + R.6;
SAD @ P.0 -2;
WOP XBUS;
WOP R.1;
SOP R.0;
SOP M.0 @ P.10;
```

Semantics:

Name	Affects	Description
FOP Fetch Operand	Address	(a) VAR and aa ← aa +/- bb (b) VAR command bits ← "read" (c) Translate[3] (d) Fetch data into MDR[4]
SAD Set Address	Address	(a) VAR and aa ← aa +/- bb (b) VAR command bits ← "store" (c) Translate[3]
SOP Store Operand	Data	(a) MDR ← aa (b) Store data from MDR[5] (Preceding CEDE must be SAD)

---

3. Translate: uses the contents of VAR as an index into translator memory and notes (internally) whether the translation is OK.
4. Fetch: if the translation is OK, initiates a fetch from memory, remembers that there is an outstanding fetch, and increments VAR by one (only the 9 low-order bits are affected; if they were all ones, then they are made zero, but there is no further carry). When the memory responds with the data, it is stored in MDR and the remembered fetch condition is cleared.
5. Store: if the (most recent) translation is OK, initiates a memory-store cycle of the word in MDR; if the translation is not OK, suppresses this ministep, and sets the PAGE action request (F.121). If the "store" command is not set in VAR, the result is undefined.

WOP Wait for Operand	Data	(a) Wait[6] (b) aa ← MDR
WOS Wait for Data, Stream Mode (privileged)	Data	(a) Wait[6] (b) aa ← MDR (c) Triggers an asynchronous mode of continuous memory fetching from successive locations in the same memory page at maximum memory rate; WOS must be executed in a loop that is faster than the memory (*viz.*, one MLP-900 cycle) lest data be lost with no indication.
ROW Retry Operation (privileged)	Address	(a) Translate[3] (b) Fetch if "read" is set in VAR[4] (Acts like FOP or SAD, depending on the old contents of VAR.)

FOP and WOP are the basic memory-fetch pair, while SAD and SOP are the basic memory-store pair. The memory currently accessed by the MLP-900 has a 750 nanosecond cycle time; allowing for translation overhead, there are at least three "free" MLP cycles available between a FOP and the following WOP.

3.2.2.3 SHIN *SH*ift *IN*struction

The SHIN ministep provides single- and double-register shifting by both fixed and variable amounts. In addition, two variants provide the basic shift-and-add steps required for multiplication and division operations. The SHIN internal format is shown in Figure 3.7. Shift codes are listed in Table 3.6 and shift amounts in Table 3.2 (see Section 3.2.2.1). The mask, shift-amount, test, A-operand, and B-operand fields (where used) are identical to those of GEAR. Indirect shift, if set, causes the encoded shift amount (although not the shift direction) to be ignored.

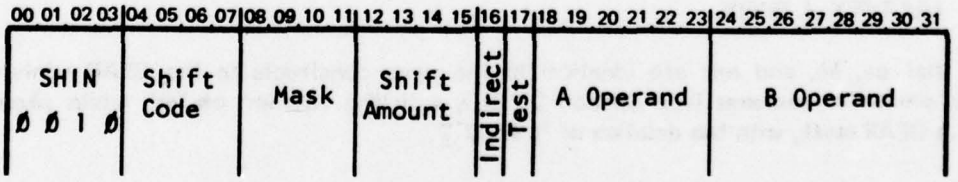

Figure 3.7 SHIN Ministep

---

6. Wait: if the last translation is not OK, suppresses this ministep and sets the PAGE action request (F.121). If there is still a memory fetch in progress, waits for it to complete (for the data to be in MDR).

Table 3.6
SHIN Shift Codes

0	SHIFT.EO.L (Shift even into odd, linear)
1	SHIFT.OE.L (Shift odd into even, linear)
2	SHIFT.SINGLE.L
3	SHIFT.DUAL.L
4	SHIFT.EO.C (Shift even and odd, circular)
5	SHIFT.RE.L (Shift register into extension, linear)
6	SHIFT.ER.L (Shift extension into register, linear)
7	SHIFT.RE.C (Shift register and extension, circular)
11	MULTIPLY
12	DIVIDE

Syntax:

{29} shin ::=
shop{30} aa{8} shdir{15} shamount{31} shmask{32} testmode{13} |
mdop{33} aa{8} BY bb{10} shmask{32} testmode{13} ;

{30} shop ::=
SHIFT.OE.L | SHIFT.EO.L | SHIFT.SINGLE.L | SHIFT.DUAL.L | SHIFT.EO.C |
SHIFT.RE.L | SHIFT.ER.L | SHIFT.RE.C

{31} shamount ::=
0 | 1 | 2 | 4 | 6 | 10 | 14 | 20 | @

{32} shmask ::=
(*M..17*) | *emptystring*

{33} mdop ::=
MULTIPLY | DIVIDE

Note that *aa*, *bb*, and *test* are identical to the same constructs in the GEAR ministep; *shamount* is similar to *samount* (see Section 3.2.2.1), with the addition of "@", while *shmask* is similar to a GEAR mask, with the deletion of "[*M..17*]".

Examples:

```
SHIFT.EO.L R.12 LEFT 6 ;
SHIFT.OE.C @P.4 RIGHT @ ;
MULTIPLY R.20 BY 12 (M.17) ;
```

Semantics:

The SHIN ministep provides for the shifting of either a single register (SHIFT.SINGLE.L), an even/odd register pair (SHIFT.EO.L, SHIFT.OE.L, SHIFT.DUAL.L, MULTIPLY, or DIVIDE), or a pair comprised of the designated register and the shift-extension register, R.37 (SHIFT.RE.L, SHIFT.ER.L, and SHIFT.RE.C). Shifting is done in two 36-bit shifters, with the designated register entering the primary shifter and the implied register entering the extension shifter; after shifting, the primary and extension shifters are copied back into the same two registers. The shift operations specify the various ways of connecting the two shifters.

aa: Designates the primary register to be shifted. For the even/odd double shifts, *aa* should be even, and the next-higher-numbered register is the implied second register of the shift; if *aa* is an odd-numbered register, then two copies of its value enter the shifter; but only the primary shifter value is stored (this allows a circular shift of a single odd register; there is no circular shift of a single even register available). For the register/extension double shifts, where R.37 is the implied register, there is no difference between an even *aa* and an odd *aa*.

shdir: The direction must be specified in the ministep as either RIGHT (/) or LEFT (\).

shamount: The shift amount (in bits) may be either direct (allowed values are the same as for GEAR) or indirect (@). Vacated bit positions are set to zero in all left shifts and to the value of SOS in all right shifts. For indirect shifts, the shift amount is taken from the shift counter, P.7; the actual shift amount is 0, 1, 2, 4, 10, or 20 (octal)--whichever is the largest value not exceeding the contents of the pointer. The pointer is decremented by the amount of the shift, and, if the new value is zero, the SHD (Shift Done) pseudo-flop is set. A paired BRAT ministep can be used to create a one-cycle shift loop to shift by an arbitrary shift amount. Note that an indirect shift cannot be paired with a BRAD ministep since the MLP cannot modify two pointers simultaneously.

shmask: The mask, if any, affects only the *aa* register itself; the implied register is always unmasked. Masked-out bits of the register enter the shifter as zero bits; their value is not altered by the shift ministep (as in the GEAR normal mode).

test: Testmode, if set, leaves all the general registers unchanged; only flops (and P.7 in an indirect shift) are affected by the execution of a test-mode SHIN.

SHIFT.SINGLE.L is a single-register shift identical to the shifting of a GEAR; this SHIN is useful only for an indirect single-register shift.

SHIFT.EO.L, SHIFT.OE.L, SHIFT.DUAL.L, SHIFT.OE.C are the straight even/odd shift operations, differing only in the connections between the two shift registers:

- -.EO.L (*E*ven-into-*O*dd *L*inear) -- bits shifted out of the even word (primary shifter) enter the odd word (extension shifter), while bits shifted out of the odd word are lost.
- -.OE.L (*O*dd-into-*E*ven *L*inear) -- bits shifted out of the even word are lost, while bits shifted out of the odd word enter the even word.
- -.DUAL.L -- bits leaving either word are lost.
- -.EO.C (*E*ven-and-*O*dd Circular) -- bits shifted out of either word enter the other one.

SHIFT.RE.L, SHIFT.ER.L, SHIFT.RE.C are the equivalent operations performed on the designated register and the extension register (R.37) as a pair:

- -.RE.L (*R*egister-into-*E*xtension *L*inear)
- -.ER.L (*E*xtension-into-*R*egister *L*inear)
- -.RE.C (*R*egister-and-*E*xtension Circular)

MULTIPLY is a single step of a multiplication loop, with the even/odd pair designating the partial product and multiplicand, respectively, and the second operand designating the multiplier. Except for timing (and, consequently, flop values) the operation "MULTIPLY X BY Y (M.Z)" is equivalent to the sequence

```
X1 ← X1 AND 1 # ! X1 is the odd reg paired with X
IF ZSP THEN, BEGIN
        X ← X + 0 (M.Z)
ELSE
        X ← X + Y (M.Z) ! add Y if LSB of X1 is set
END ;
SHIFT.EO.L X RIGHT 1 (M.Z) ;
```

DIVIDE is a single step of a division loop, with the first operand (even/odd pair) designating the dividend (which develops into quotient and remainder) and the second operand designating the divisor. Except for timing, the operation "DIVIDE X BY Y (M.Z)" is equivalent to the sequence

```
IF COF.1 THEN.BEGIN ! the current setting selects ...
        X ← X - Y (M.Z) ! ... either subtraction ...
ELSE
        X ← X + Y (M.Z) ! ... or addition
END ;
SHIFT.OE.L X LEFT 1 (M.Z) ;
IF COF.1 THEN.BEGIN ! the new setting (from above) ...
        X1 ← X1 OR 1 ! ... is the new quotient bit in X1
END ;
```

Note that COF.1 must be properly initialized for a divide loop; subsequent iterations use the value set by the previous iteration.

The following flops are used uniformly in all SHIN ministeps:

SOS--on all right shifts (including MULTIPLY) a copy of SOS is brought into vacated bit positions: into the unconnected register in a linear shift or into both registers in the dual shift. SOS is not used in a circular shift.

SHE--on all linear left shifts SHE is set to the value of the last bit shifted out of the unconnected register. SHE is not affected by circular or dual shifts.

SOF--on all linear left shifts SOF is set if any bit shifted out of the unconnected register is different from the setting of SOS. SOF is never cleared by a shift. SOF is not affected by circular or dual shifts.

SHD--pseduo-flop that is valid only during an indirect-shift cycle.

The following flops are associated with the adder, and are affected only by the MULTIPLY and DIVIDE operations:

COP, COF.1--reflect the carry-out of the adder (COP is valid only *during* this cycle; COF.1 is valid only *after* this cycle). COF.1 is also an input to DIVIDE.

COF.2--at the end of this cycle, contains the value of COF.1 from the beginning of this cycle.

3.2.2.4 GENT *GEN*eral Data Transfer

This ministep performs data transfers between OE registers and is also used, in conjunction with the CE ministep MOVE, to provide interengine data transfers. The GENT internal coding is shown in Figure 3.8. GENT takes two operands: A and B. The direction of the transfer is controlled by the To/From bit:

To/From	Result
0	A ← B
1	B ← A

The 12-bit address for the A-operand is coded in three sections as described for CEDE in Section 3.2.2.2. If the A-operand addresses the mask registers, or if the destination is an immediate value or a pointer register, the resulting operation is a no-op. The B-operand is coded as described in Section 3.2.1, except that the "Op B Group" field is used when bits 0 and 1 are zero; otherwise, the "Op B Group" field must be zero. The registers addressed by the "Op B Group" field are shown in Table 3.7.

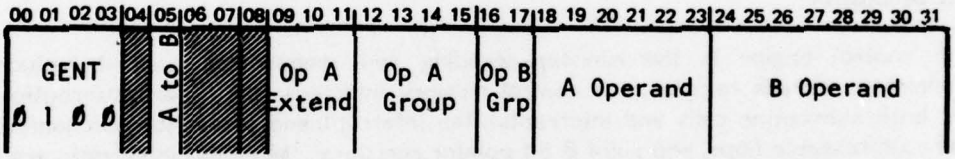

Figure 3.8 GENT Ministep

Table 3.7
GENT B Operand Group

Op B Group	Register
0	Normal B-operand (see Section 3.2.1)
1	*M..17* -- Mask Registers
2	*MISC...37* -- Misc. Registers
3	XBUS -- Exchange Bus

Syntax:

```
{34} gent ::=
        genta{35} ← gentb{37} ; | genta{35} ← gentc{39} ; | gentb{37} ← genta{35} ;

{35} genta ::=
        gentar{36} | gentar{36} @ P..7 | gentar{36} * P..7 | XBUS

{36} gentar ::=
        R..37 | MISC...37 | A..1777 | XLATOR..777

{37} gentb ::=
        gentbr{38} | gentbr{38} @ P..7 | gentbr{38} * P..7 | XBUS
```

{38} gentbr ::=
R..37 | M..17 | MISC..37

{39} gentc ::=
number{5} | *P..7*

Examples:

```
R.12 ← 1234567 ;
MISC.12 ← XBUS ;
A.123 ← P.12 ;
XBUS ← CE.0 ;
XBUS ← A.1234 ;
```

Semantics:

GENT performs transfers among the OE registers (see Table 3.1). The contents of the right register is copied into the left register. Where XBUS is used as a destination (left register) or a source (right register), the GENT should be paired with a corresponding MOVE to transfer data between the CE and OE.

3.3 Control Engine

The control engine is the ministep-decoding and -sequencing unit; it includes the current-ministep address register, the control memory interface, a 16-word subroutine stack (used for both subroutine calls and interrupts), the interrupt and protection mechanisms, 256 individually addressable flops, and eight 8-bit pointer registers. MLP-900 interrupts are known as "action requests" (AR's). There are 32 AR levels, of which 24 are privileged. Of the eight levels available to user microcode, only two have dedicated functions in PRIM (see Section 2.2.1); the others can be defined by the user. CE ministeps allow conditional branching (including subroutine calls and returns) and simple flop and pointer-register computations.

3.3.1 Control Engine Operands

CE Registers. A CE byte (register) consists of a 4-bit group number and a 4-bit register-within-group number. This encoding is shown in Figure 3.9.

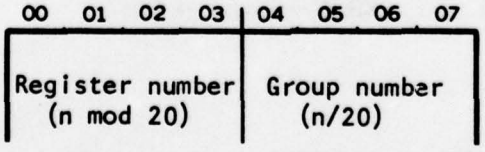

Figure 3.9 CE.n Encoding

Relative Addresses. A relative address is encoded in one byte; it is relative to the continuation address (the next instruction word). Thus a skip is coded as +1 instead of +2. The relative offset is a signed, two's-complement value in the range -200 through +177, octal. In GPM all relative addresses are specified relative to the current instruction (or through a lable); because the encoded offset is relative to the continuation address, however, the effective range for relative addresses in GPM is -177 through +200, octal.

Flop Expressions. A flop expression is encoded in two and one-half bytes. Two bytes contain the flops encoded as shown in Figure 3.11. The half-byte defines the function. Figure 3.10 shows where this information is placed in the instruction word. A flop and its associated true bit are used in BRAT, BENT, BORE, BRAD, BEAD, and MAST ministeps to form flop terms. If the true bit is set, then the actual flop value is used; if it is off, then the flop's complement is used.

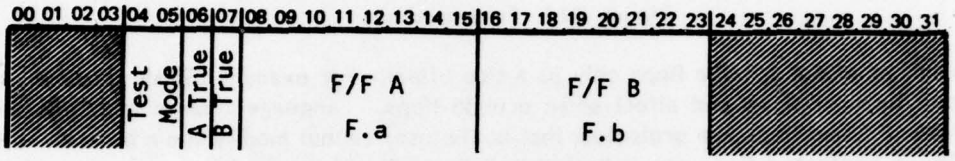

Figure 3.10 Boolean Expression Encoding

Table 3.8
Boolean Expression Types

Test Mode	A True	B True	Boolean Expression
00	0	0	F.b ← NOT F.a
		1	NOT (F.b ← F.a)
	1	0	NOT (F.b ← NOT F.a)
		1	F.b ← F.a
01	0	0	NOT F.b OR NOT F.a
		1	F.b OR NOT F.a
	1	0	NOT F.b OR F.a
		1	F.b OR F.a
10	0	0	NOT F.b AND NOT F.a
		1	F.b AND NOT F.a
	1	0	NOT F.b AND F.a
		1	F.b AND F.a
11	0	0	NOT F.b XOR NOT F.a
		1	F.b XOR NOT F.a
	1	0	NOT F.b XOR F.a
		1	F.b XOR F.a

3.3.1.1 *F..377* Flip-Flops

CE..37 represents 32 bytes of addressable flops, known individually as *F..377*, that may be set and tested directly by most of the CE ministeps. Within a byte, flops are ordered from high- to low-order bit. Flops are organized into two major functional groups: F.0-F.277 are real flops; F.300-F.377 are pseudo-flops. For encoding purposes, the flops are divided into two groups. F.0-F.177 are all in group 0, and F.200-F.377 are all in group 1. Thus F.327 is coded as flop number 127 in group 1. This encoding is shown in Figure 3.11.

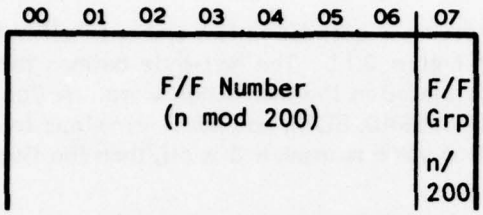

Figure 3.11 F.n Encoding

Some ministeps affect specific flops only as a side effect. For example, GEAR and SHIN use and modify one byte of flops and affect some pseudo-flops. Language boards and AR's also use certain flops. Some flops are protected; that is, the user cannot modify them but can reference them. These protected flops are indicated in Table 3.9 and the text below by an asterisk (*) beside the flop name.

Table 3.9 lists all the flops. The flop number is the sum of the numbers at the top of the column and in the extreme left row in which the flop is located. Where the flop number appears (*e.g.*, F.135) rather than a mnemonic, the flop is unassigned; where three dashes (---) appear, the flop is unimplemented. The pseudo-flops in CE.30 (F.300-F.307), plus SHD (F.353), reflect conditions that arise in the current cycle and are defined only when the appropriate ministeps are being executed; ***all other flops*** reflect conditions as of the beginning of the current cycle. A reference to any flop in CE.30 causes a one-cycle "hiccup"; the cycle requires two clock periods to execute. The flops in CE.30 cannot be referenced in CALL or RETURN ministeps. The following are real flops:

F..57 General Indicators: available to user microcode for arbitrary usage.
* ***SLBC..17*** (F.60-F.77) Supervisor Language Board Controls.
* POWER, PANIC, OPAR, ... (F.100-F.127) Action Requests.
TRAC, ... (F.130-F.137) User-level AR's: Each flop represents a specific pending AR that causes a microcode interrupt whenever its appropriate level is enabled. Each bit can be set either by the specific occurrence it represents or by a ministep.[7]
COF.1, COF.2, ZRF.1, ZRF.2, SHE, SOS, SOF (F.140-F.147) Carryout flops, zero flops, shift extension, shift-out sign, shift-out flag: OE-associated (GEAR and SHIN) flops; fully described in the GEAR and SHIN sections.
ARL.5 (F.150) AR Lockout: user-level AR lockout.[7]
ITRAC (F.153) Initiate Trace.[7]
F.154-F.157 General Indicators: available to user microcode for arbitrary usage.
* ***SARM..1*** (F.160-F.161) Supervisor AR Masks: control the memory-compare AR.
* CKC (F.164) Clock Control.
* TRBY (F.165) Translator Bypass.
* CKT (F.166) Check Test.
* MBS (F.167) Mask Bank Selector: selects current mask bank.
* ARL.1-4 (F.170-F.173) AR Lockout: lockouts for privileged AR levels.[7]
* ***MOD..1*** (F.174-F.175) Mode Bits: stored in control memory by a BLOT WCM.
* SUPVLB (F.176) Supervisor LB: selects Supervisor LB.
* SUPVCT (F.177) Supervisor Control: forces MLP-900 into supervisor mode regardless of the mode bit in control memory.
F.200-F.237 General Indicators: available to user microcode for arbitrary usage.

---

7. See Section 3.3.3 on AR's.

The following are pseudo-flops.

COP (F.300) Carry-out Pseudoflop: see GEAR (Section 3.2.2.1) and SHIN (Section 3.2.2.3) instructions.
ZSP (F.301) Zero-sense Pseudoflop: see GEAR instruction (Section 3.2.2.1).
THZ (F.304) Through Zero: see BRAD instruction (Section 3.2.3.1).
WAR (F.305) Wait AR: one of F.133-F.137 is pending.
CCP (F.307) Check-Carry Pseudoflop: carryout *from* the check-adder.
TRUE (F.310): always set.
SSW..7 (F.340-F.347) Sense Switches: from the MLP control panels.
SHD (F.353) Shift Done: see SHIN instruction (Section 3.2.2.3).
OSI..3 (F.354-F.357) One-sense Indicate: senses -1 in the corresponding *P..3*.
ZSI..7 (F.360-F.367) Zero-sense Indicate: senses 0 in the corresponding *P..7*.
TSI..1 (F.374-F.375) Three-sense Indicate: senses 3 in the corresponding *P..1*.
FSI..1 (F.376-F.377) Four-sense Indicate: senses 4 in the corresponding *P..1*.

3.3.1.2 *P..7* Pointer Registers

There are eight 8-bit pointer registers that can be used in the OE to address registers indirectly (*e.g.*, R.0 @ P.3 is the general register determined by the low-order 5 bits of P.3). The pointer registers can be loaded by a MOVE instruction, modified by the BRAD instruction, and tested through the pointer-sense pseudo-flops. The following pointers have special-purpose functions:

P.0-P.3	used and modified by the BLOT ministep; otherwise generally available.
P.6	stack pointer (dedicated for micro-PC).
P.7	shift counter for SHIN.

The following pseudo-flops are true if, and only if, the appropriate pointer has exactly the specified value.

OSI..3	sense all ones (*i.e.*, -1 or octal 377) in the corresponding *P..3*.
ZSI..7	sense zero (0) in the corresponding *P..7*.
TSI..1	sense the value three (3) in the corresponding *P..1*.
FSI..1	sense the value four (4) in the corresponding *P..1*.

When a BRAD ministep both modifies a pointer and tests that pointer's sense pseudo-flops, the original value of the pointer is sensed.

Table 3.9
Flip-Flops (Names and Groups)

	F.0	F.40	F.100	F.140
	(CE.0)	(CE.4)	(CE.10)	(CE.14)
00	F.0	F.40	POWER*	COF.1
01	F.1	F.41	PANIC*	.2
02	F.2	F.42	OPAR*	ZRF.1
03	F.3	F.43	EPAR*	.2
04	F.4	F.44	SOVF*	F.144
05	F.5	F.45	SUNF*	SHE
06	F.6	F.46	UOVF*	SOS
07	F.7	F.47	UUNF*	SOF
	(CE.1)	(CE.5)	(CE.11)	(CE.15)
10	F.10	F.50	CMADR*	ARL.5
11	F.11	F.51	AERR*	F.152
12	F.12	F.52	BERR*	F.153
13	F.13	F.53	PERR*	ITRAC
14	F.14	F.54	F.114*	F.154
15	F.15	F.55	F.115*	F.155
16	F.16	F.56	MMERR*	F.156
17	F.17	F.57	F.117*	F.157
	(CE.2)	(CE.6)	(CE.12)	(CE.16)
20	F.20	SLBC.0*	TASK*	SARM.0*
21	F.21	.1*	PAGE*	.1*
22	F.22	.2*	SUPVF*	F.162*
23	F.23	.3*	PROT*	F.163*
24	F.24	.4*	VADR*	CKC*
25	F.25	.5*	F.125*	TRBY*
26	F.26	.6*	F.126*	CKT*
27	F.27	.7*	F.127*	MBS*
	(CE.3)	(CE.7)	(CE.13)	(CE.17)
30	F.30	SLBC.10*	TRAC	ARL.1*
31	F.31	.11*	F.131	.2*
32	F.32	.12*	F.132	.3*
33	F.33	.13*	F.133	.4*
34	F.34	.14*	F.134	MOD.0*
35	F.35	.15*	F.135	.1*
36	F.36	.16*	F.136	SUPVLB*
37	F.37	.17*	F.137	SUPVCT*

Table 3.9 (Continued)

	F.200	F.240	F.300	F.340
	(CE.20)	(CE.24)	(CE.30)	(CE.34)
00	F.200	F.240	COP[8]	SSW.0
01	F.201	F.241	ZSP[8]	.1
02	F.202	F.242	---	.2
03	F.203	F.243	---	.3
04	F.204	F.244	THZ[8]	.4
05	F.205	F.245	WAR	.5
06	F.206	F.246	---	.6
07	F.207	F.247	CCP[8]	.7
	(CE.21)	(CE.25)	(CE.31)	(CE.35)
10	F.210	F.250	TRUE	---
11	F.211	F.251	---	---
12	F.212	F.252	---	---
13	F.213	F.253	---	SHD[8]
14	F.214	F.254	---	OSI.0
15	F.215	F.255	---	.1
16	F.216	F.256	---	.2
17	F.217	F.257	---	.3
	(CE.22)	(CE.26)	(CE.32)	(CE.36)
20	F.220	F.260	---	ZSI.0
21	F.221	F.261	---	.1
22	F.222	F.262	---	.2
23	F.223	F.263	---	.3
24	F.224	F.264	---	.4
25	F.225	F.265	---	.5
26	F.226	F.266	---	.6
27	F.227	F.267	---	.7
	(CE.23)	(CE.27)	(CE.33)	(CE.37)
30	F.230	F.270	---	---
31	F.231	F.271	---	---
32	F.232	F.272	---	---
33	F.233	F.273	---	---
34	F.234	F.274	---	TSI.0
35	F.235	F.275	---	.1
36	F.236	F.276	---	FSI.0
37	F.237	F.277	---	.1

8. Reflects conditions only within the current cycle.

3.3.1.3 *CE..77* Miscellaneous Registers

The double register pair (CE.60, CE.61) is the miniflow status word, of which only 2 bits are used.

Figure 3.12 Miniflow Status Word

LB selects the active language board set.

The double register pair (CE.62, CE.63) is the current address register. It contains the address of the current instruction or of the first instruction of a pair. A MOVE to the current address register is a no-op.

CE.64-CE.67 comprise the exchange bus from the OE into the CE; it is addressed as *XBUS..3* on the left side of an assignment in the MOVE ministep. CE.70-CE.73 comprise the exchange bus from the CE into the OE; it is addressed as *XBUS..3* on the right side of the assignment in the MOVE ministep. *XBUS..3* are pseudo-registers connected to bits 4-35 of the exchange bus in the OE: XBUS.0 connects to bits 4-11, XBUS.1 to bits 12-19, XBUS.2 to bits 20-27, and XBUS.3 to bits 28-35.

3.3.1.4 *S..17* Subroutine Stack

The Subroutine Stack consists of sixteen 16-bit registers. The subroutine stack, together with P.6 (the stack pointer), is automatically used in AR's and in subroutine calls and returns. A subroutine call (a BEAD or BENT ministep) branches to the effective address and pushes the return address onto the top of the stack. This is done by incrementing P.6 by 1 and then using the four low-order bits to select the stack word to be loaded with the return address. In addition, if the four low-order bits of P.6 were octal 16 (indicating that the stack is being filled), either a supervisor stack overflow (F.104) or a user stack overflow (F.106) is requested, according to the mode of the caller. Taking an AR consists of pushing the interrupted address onto the stack and branching to the AR entry point, simultaneously setting the appropriate lockout bit (ARL.1-ARL.5).

A return (*i.e.*, a BORE ministep) loads the current address register from the top of the stack and then decrements P.6 by 1. If the stack is empty (the four low-order bits of P.6 are 0) and if ARL.2 is off, a stack underflow of the appropriate kind is taken (F.105 if supervisor; F.107 if user), the pointer is left unchanged, and the current address (*i.e.*, the address of the BORE instruction) is stacked in S.0. If the stack is empty but ARL.2 is on, the BORE returns normally, decrementing P.6 as it goes.

3.3.2 Control Engine Operators

The CE operators are:

- BRAT Branch with Test -- provides conditional jumps.
- BENT Branch and Enter -- provides conditional subroutine calls.
- BORE Branch or Return -- provides conditional subroutine returns.
- BRAD Branch and Modify -- provides loop control.
- BEAD Branch Extended Address -- provides conditional and unconditional subroutine calls and jumps. It has a larger addressing capability than BRAT or BENT.
- BLOT Block Transfer -- provides loop control together with data transfers within the OE.
- MAST Manipulate Status -- manipulates flops.
- MOVE Move CE Registers -- the general data transfer instruction for the CE.

3.3.2.1 BRAT *BRAnch with Test*

The BRAT internal coding, given in Figure 3.13, consists of the BRAT opcode, a boolean expression, and a relative address (see Section 3.3.1, Figure 3.10, and Table 3.8).

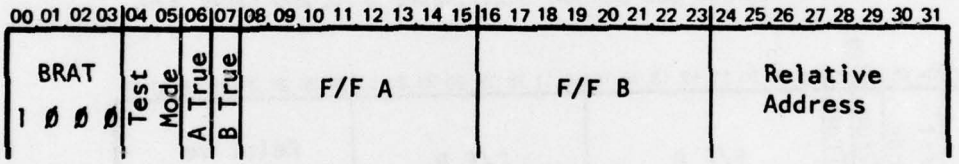

Figure 3.13 BRAT Ministep

Syntax:

```
{40}  brat ::=
          / IF flopexp{41} THEN GOTO rellabel{44} END ;

{41}  flopexp ::=
          flopterm{42} bop{43} flopterm{42}  |  ( F..277 ← flopterm{42} )  |
          NOT ( F..277 ← flopterm{42} )

{42}  flopterm ::=
          NOT F..377  |  F..377  |  FALSE
          <<FALSE is a shorthand for NOT TRUE>>

{43}  bop ::=
          AND | OR | XOR

{44}  rellabel ::=
          offset{45}  |  id{1}

{45}  offset ::=
          + number{5}  |  - number{5}
```

Examples:

```
/IF (F.0 ← TRUE) THEN GOTO +200 END;
/IF NOT (F.1 ← FALSE) THEN GOTO -177 END;
/IF F.3 OR F.3 THEN GOTO TAG17 END;
/IF NOT F.4 AND F.5 THEN GOTO +7 END;
/IF F.377 XOR NOT F.377 THEN GOTO -3 END;
/IF NOT F.1 OR NOT F.4 THEN GOTO +166 END;
```

Semantics:

This ministep provides conditional jumps. If the boolean expression ***flopexp*** evaluates to true, then the branch is taken; otherwise execution continues with the next instruction. If a store (←) is specified in the boolean expression, the store occurs whether the branch is taken or not. The branch destination is a location relative to the current instruction. The limits on the branch destination are octal +200 and -177, inclusive. As with all relative branches, addressing beyond or before the ends of control memory will cause a location-counter wraparound. Thus a transfer to +70 from location 7747 will go to location 0037.

3.3.2.2 BENT *Branch and ENTer*

The BENT internal coding, given in Figure 3.14, consists of the BENT opcode, a boolean expression and a relative address (see Section 3.3.1, Figure 3.10, and Table 3.8).

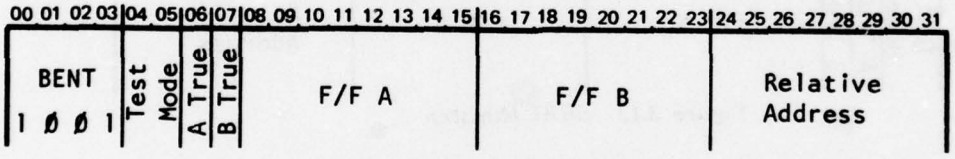

Figure 3.14 BENT Ministep

Syntax:

{46} bent ::=
/ IF flopexp{41} THEN CALL rellabel{44} END ;

Examples:

```
/IF (F.17 ← NOT F.1) THEN CALL SUB END;
/IF F.202 OR F.206 THEN CALL +1 END;
/IF F.4 XOR NOT F.77 THEN CALL -27 END;
```

Semantics:

This ministep provides conditional subroutine calls. The execution of the BENT ministep is similiar to the BRAT. The only difference is that when the branch is taken, a subroutine entry is executed, with the address of the next instruction being pushed onto the subroutine stack (*S..17*).

3.3.2.3 BORE *Branch Or REturn*

The BORE internal coding, given in Figure 3.15, consists of the BORE opcode, a boolean expression and a relative address (see Section 3.3.1, Figure 3.10, and Table 3.8).

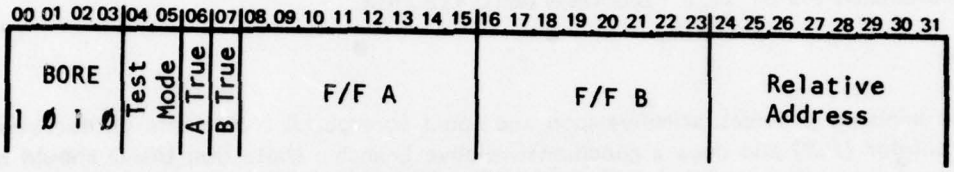

Figure 3.15 BORE Ministep

Syntax:

{47} bore ::=
/ IF flopexp{41} THEN GOTO rellabel{44} ELSE RETURN END ;

Examples:
/IF F.1 OR NOT F.3 THEN GOTO -3 ELSE RETURN END;
/IF TRUE OR F.0 THEN GOTO +1 ELSE RETURN END;

Semantics:

This ministep provides conditional subroutine returns (there is no unconditional subroutine return). The execution of the ministep is identical to BRAT if the boolean expression evaluates to true. If the expression evaluates to false, then instead of continuing at the next instruction, a subroutine return is executed. As with both BRAT and BENT, if a store is indicated, it occurs whether the expression evaluates to true or false.

3.3.2.4 BRAD *BRanch And moDify pointer*

The BRAD internal coding, given in Figure 3.16, consists of the BRAD opcode, a pointer register number, a modifier (the pointer's increment/decrement), a flop term (which corresponds to the B-part of a boolean expression), and a relative address (see Section 3.3.1, Figure 3.10, and Table 3.8).

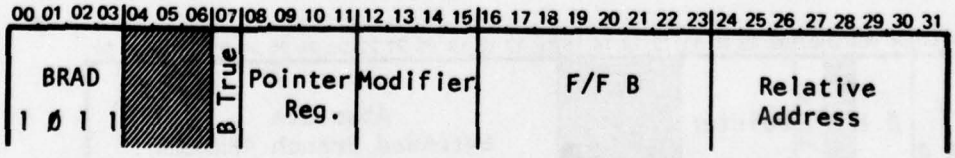

Figure 3.16 BRAD Ministep

Syntax:

{48} brad ::=
/ bradop{49} *P*..7 BY number{5} ; IF flopterm{42} THEN GOTO rellabel{44} END ;

{49} bradop ::=
INCREMENT | DECREMENT

Examples:
/INCREMENT P.1 BY 7; IF TSI.1 THEN GOTO TAG53 END;
/DECREMENT P.0 BY 10; IF FSI.0 THEN GOTO +12 END;

Semantics:

This ministep provides primitive loop and count control. It increments or decrements a counting pointer (*P..7*) and does a conditional relative branch. (Note that BRAD should *not* be executed in a pair with a SHIN ministep using indirect shift.) The largest increment is 7 and the largest decrement is octal 10. The through-zero (THZ) pseudo-flop is defined only for a BRAD ministep; whenever the pointer value (taken as an 8-bit, non-negative number) overflows or underflows, THZ is true and the new pointer value is correct modulo 400 (octal).

3.3.2.5 BEAD *Branch Extended ADdress*

The BEAD instruction provides for both conditional branching to any location in control memory and unconditional indexed branching using a pointer. There are four forms of BEAD, with syntax for all given in Section 3.3.2.5. They may each be used as a CALL or a GOTO, as determined by the "Enter" bit shown in Figures 3.17-20: if "Enter" is set, a CALL occurs rather than a GOTO.

BEAD0. The BEAD0 internal coding, given in Figure 3.17, consists of a BEAD0 opcode, a flop term (see Section 3.3.1), and a 16-bit absolute address.

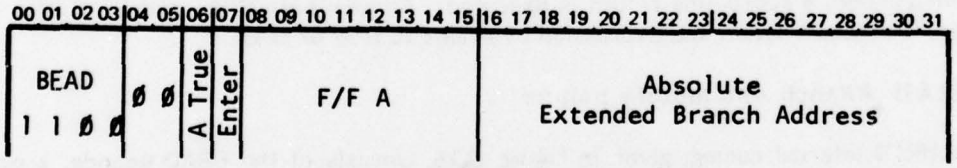

Figure 3.17 BEAD0 Ministep

BEAD1. The BEAD1 internal coding, given in Figure 3.18, consists of a BEAD1 opcode, a pointer register number, and a 16-bit absolute address.

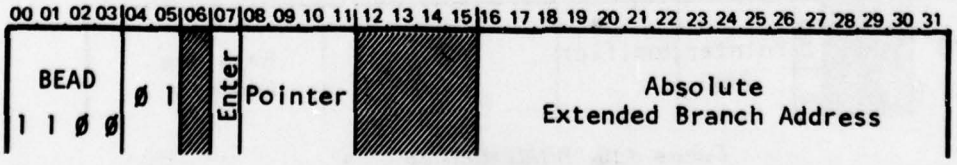

Figure 3.18 BEAD1 MInistep

BEAD2. The BEAD2 internal coding, given in Figure 3.19, consists of a BEAD2 opcode and a pointer register number.

Figure 3.19 BEAD2 Ministep

BEAD3. The BEAD3 internal coding, given in Figure 3.20, consists of a BEAD3 opcode, a flop term (see Section 3.3.1), and a 16-bit two's-complement relative address (relative to the next instruction).

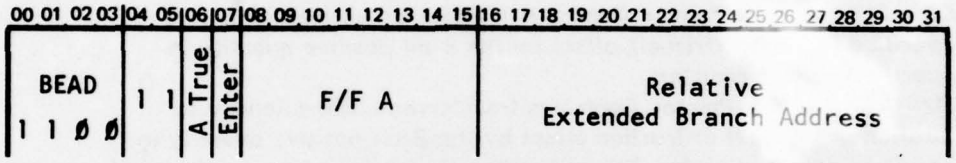

Figure 3.20 BEAD3 Ministep

Syntax:

```
{50}  bead ::=
          bead0{51} | bead1{52} | bead2{53} | bead3{54}

{51}  bead0 ::=
          / IF flopterm{42} THEN trfrop{55} trfrlabel{56} END ;

{52}  bead1 ::=
          / trfrop{55} trfrlabel{56} < P..7 > ;

{53}  bead2 ::=
          / trfrop{55} +1 < P..7 > ;

{54}  bead3 ::=
          / IF flopterm{42} THEN trfrop{55} sign{22} number{5} END ;

{55}  trfrop ::=
          CALL | GOTO

{56}  trfrlabel ::=
          number{5} | id{1}
```

Examples:

```
/IF F.1 THEN GOTO TAG67 END;
/IF NOT F.13 THEN CALL 200 END;
/CALL TAG31 <P.57>;
/GOTO 277 <P.11>;
/CALL +1 <P.4>;
/GOTO +1 <P.11>;
/IF TRUE THEN GOTO +3711 END;
```

/IF NOT F.11 THEN GOTO -67 END;

Semantics:

This ministep provides unconditional or indexed jumps or subroutine calls. The major function of the BEAD, however, is to provide extended branch-addressing capability. BEAD is the only ministep that can transfer beyond the relative address range -200 through +177 (octal) since it can address all of control memory. All BEADs may optionally execute a subroutine call. There are four forms of BEAD ministeps:

- BEAD0 - Conditional Absolute. If the specified flop expression (see Section 3.3.1) is true, control is transferred absolutely to any location (trfrlabel) in control memory.
- BEAD1 - Absolute plus Pointer. Control is transferred unconditionally to the specified location (trfrlabel), offset by the 8-bit positive quantity in the specified pointer register.
- BEAD2 - Relative plus Pointer. Control is transferred unconditionally to the location of the next instruction offset by the 8-bit positive quantity in the specified pointer register. This instruction always transfers in a forward direction.
- BEAD3 - Conditional Relative. If the specified flop expression (see Section 3.3.1) is true, control is transferred to the location of the next instruction offset by a 16-bit two's complement displacement.

3.3.2.6 BLOT *BLOck Transfer*

The BLOT internal coding, given in Figure 3.21, consists of the BLOT code and a relative address (see Section 3.3.1). BLOT codes are given in Table 3.10.

Table 3.10
BLOT Codes

0	RCM	Read Control Memory*
1	WCM	Write Control Memory*
2	RSB	Read Subroutine Stack
3	WSB	Write Subroutine Stack
4	MOE	Move OE
5	WBP	Write Contol Memory, Bad Parity*

* Indicates a privileged code.

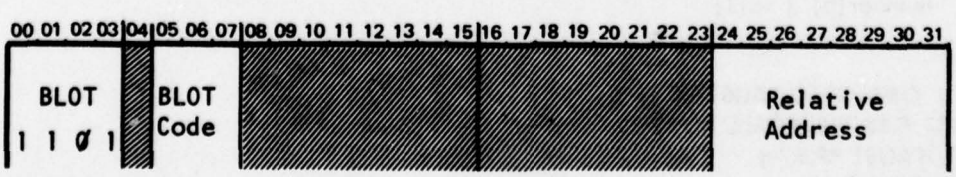

Figure 3.21 BLOT Ministep

Syntax:

{57} blot ::=
blotcode{58} rellabel{44} ;

{58} blotcode ::=
RCM | WCM | RSB | WSB | MOE | WBP

Examples:
RCM +7;
WBP -5;

Semantics:

BLOT is used to establish loops to transfer blocks of data. The execution of a single BLOT ministep can simultaneously move one word of data, modify some pointers, and conditionally branch. There are six types of BLOTs--one facilitates moving data in the OE, two reference the subroutine return stack, and three reference control memory (the only instructions that do so). Three steps occur simultaneously in all types of BLOT transfers:

(1) Move CE data to or from the XBUS, as specified by the BLOT type.

(2) Modify Pointers. Pointer register modification is identical for all six types of block transfers: P.0 and (P.2, P.3) as a single 16-bit register are each incremented by one, and P.1 is decremented by one. Note that the data-move and conditional-branch parts of the BLOT, plus any paired OE ministep, use the old values of the pointer registers.

(3) Conditional Branch. The conditional branch function is identical for all six types of block transfer. Each time BLOT is executed, P.1 (the word counter) is tested. When a count of one is present, execution continues with the next instruction. If P.1 contains any count other than one, control is transferred to the branch address. A word count of zero initially loaded into P.1 may be used to transfer a block of 256 words.

The data transfer functions for the various BLOTS are:

MOE: No CE data is moved, but steps 2 and 3 above are performed.

RSB: Read one word from Subroutine Stack into XBUS (XBUS.2, XBUS.3).
WSB: Write one word into Subroutine Stack from XBUS.
These two BLOTs read and write subroutine-stack words. They are 16 bits wide and read from or write to the rightmost 16 bits (*i.e.*, H.1) of XBUS. The low-order four bits of P.3 select the stack word (*S..17*).

RCM: Read one word from control memory into XBUS.
WCM: Write one word into control memory from XBUS with good parity.
WBP: Write one word into control memory from XBUS with bad parity.
These three privileged BLOTs are the only instructions that can reference control memory. They are 36 bits wide, reading and writing via the XBUS and using (P.2, P.3) to select the control-memory address. A control-memory word

is 40 bits wide: thirty-six data or instruction bits come from the XBUS; two mode bits come from flops MOD.0 (F.174) and MOD.1 (F.175); one bit is a parity bit, either good or bad; and one is unused and is always 0. Parity is generated automatically. WCM generates odd (good) parity; WBP generates even (bad) parity. RCM and WBP are used in diagnostics. WCM is used for swapping in a new user.

3.3.2.7 MAST *M*anipulate *ST*atus

The MAST internal coding, given in Figure 3.22, consists of a MAST opcode, a logical function, two flop terms (see Section 3.3.1), and a result flop. The MAST logical functions are given in Table 3.11; the operand notation follows Figure 3.22.

Table 3.11
MAST Logical Codes

0	IF F.b.term THEN F.result ← F.a.term
1	F.result ← F.a.term OR F.b.term
2	F.result ← F.a.term AND F.b.term
3	F.result ← F.a.term XOR F.b.term

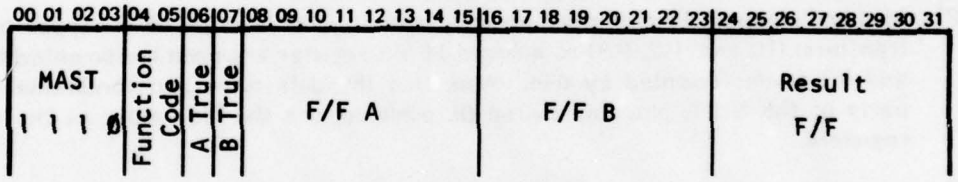

Figure 3.22 MAST Ministep

Syntax:

{59} mast ::=
F..277 ← flopterm{42} bop{43} flopterm{42} ; |
/ IF flopterm{42} THEN *F..277* ← flopterm{42} END ;

Examples:

F.1 ← F.17 OR NOT F.20 ;
F.33 ← NOT F.106 XOR F.13 ;
F.106 ← TRUE OR TRUE ;
/IF F.6 THEN F.111 ← NOT F.4 END ;
/IF NOT F.11 THEN F.4 ← F.22 END ;

Semantics:

This ministep manipulates flops. There are two types of MAST ministeps, the unconditional and conditional store.

Unconditional MAST. This form of MAST stores a two-term boolean expression into a third flop. A flop may be referenced several times. For example, the following will complement F.7:

```
F.7 ← NOT F.7 OR NOT F.7 ;
```

Conditional MAST. If the term being tested is true, a store is made. For example, the following two MAST statements have the same result:

```
/IF NOT F.7 THEN F.7 ← NOT F.10 END ;
F.7 ← F.7 OR NOT F.10 ;
```

3.3.2.8 MOVE *MOVE* CE Registers

The MOVE internal coding, given in Figure 3.23, consists of a from-address, a to-address, and an immediate mask. The from-address is a constant in the case of Move-Immediate; a flop in the case of the Move-Flop; and a CE register for the other four MOVE's. The to-address is always a CE register. The immediate mask is an 8-bit constant; it is not used in the double-byte MOVE. The MOVE codes are given in Table 3.12.

Table 3.12
MOVE Codes

0	MSI	Move Immediate
1	MOM	Move Flop
2	MAR	Move Register
3	MAC	Move and Complement
4	MCL	Move and Clear
5	MDB	Move Double Byte

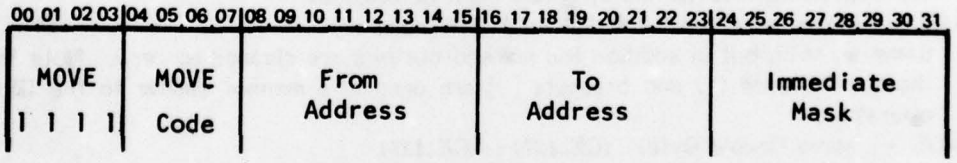

Figure 3.23 MOVE Ministep

Syntax:

```
{60}  move ::=
          CE..137 ← msingle{61} ; | ( CE..137 ) ← ( CE..137 ) ;

{61}  msingle ::=
          msource{62} | msource{62} ( number{5} ) | CE..137 [ number{5} ]

{62}  msource ::=
          number{5} | F..377 | CE..137 | NOT CE..137
```

Examples:

```
CE.17 ← 5 (7) ;
P.0 ← 17 (75) ;
CE.111 ← F.113 (355) ;
GOR.1 ← G1R.3 (377) ;
```

```
XBUS.3 ← NOT CE.4 ;
CE.4 ← XBUS.0 [174] ;
(CE.1) ← (CE.0) ;
S.2 ← (P.0) ;
```

Semantics:

This ministep provides data transfer between CE registers; it is also used in conjunction with the OE ministep GENT to provide interengine data transfers. There are six types of MOVE ministeps. All but one of these set one CE register, making use of an immediate mask value specified in parentheses or brackets. If the mask is not specified, a mask of all one-bits, (377), is assumed. The mask value is similar to the mask register used in the OE; only bits corresponding to ones in the mask are modified. Note that in a MOVE to the exchange bus, the mask is ignored and the entire byte is moved. The double MOVE copies an even/odd register pair to another even/odd register pair; the mask is not used.

- MSI -- Move Immediate: *CE..137* ← number (number).
  All masked-in bits of the left CE register receive the corresponding value of the specified constant right operand. As in the GEAR, the mask is specified in parentheses ( ).
- MOM -- Move Flop: *CE..137* ← *F..377* (number).
  All masked-in bits of the left CE register receive the value of the specified flop.
- MAR -- MOVE: *CE..137* ← *CE..137* (number).
  All masked-in bits of the left CE register receive the corresponding value of the specified right CE register.
- MAC -- Move Complemented: *CE..137* ← NOT *CE..137* (number).
  All masked-in bits of the left CE register receieive the complement of the corresponding value of the specified right CE Register.
- MCL -- Move and Clear: *CE..137* ← *CE..137* [number].
  Same as MAR, but in addition the masked-out bits are cleared to zero. Note that the parentheses ( ) and brackets [ ] are used in a manner similar to the GEAR operation.
- MDB -- Move Double Byte: (*CE..137*) ← (*CE..137*).
  Moves one pair of CE registers to another pair of CE registers. The pairs are always an even/odd register pair. Thus (CE.4) and (CE.5) both specify the pair (CE.4, CE.5). When both registers specified are even or both odd, the MOVE will be normal, that is, even to even and odd to odd. When the specified registers are one even and one odd, however, the MOVE will be reversed, that is, even to odd and odd to even. *S..17* can be used to represent the appropriate even/odd CE register-pair.

3.3.3 Action Requests

There are 32 action-request (AR) flops (F.100-F.137). Each one is connected to an interrupt location (see the Address column in Table 3.13); in addition, each AR is associated with one of five lockout levels (ARL.1-ARL.5). ARL.1 locks out all ARs; ARL.2 all ARs on levels 2-5, etc.

When the CE senses the existence of an immediate AR that is not locked out, the current clock cycle is inhibited (*i.e.*, the current ministep is suppressed) and in the next cycle the MLP-900 takes the AR by performing a call (using the stack to store the interrupted address)

to the AR entry point, simultaneously setting the lockout bit of the interrupt level being entered. For those ARs of type "Wait" (see Table 3.13), the AR remains pending until the next WOP instruction, when the AR takes place (if not locked out by a higher level). Since the AR flops are not turned off by the interrupt itself, they must be turned off by the interrupt routine.

There are eight action-request (AR) levels available to the user microcode: three immediate and five wait. Of these eight, only TRAC has an assigned function: a user trace function is implemented through the TRAC AR and the ITRAC flop. One cycle after the ITRAC flop is set (by microcode), the MLP-900 sets TRAC and clears ITRAC. Thus a TRAC AR routine of the form

```
TRAC ← FALSE;
   . . .        !trace conditions
ARL.5 ← FALSE;
IF (ITRAC ← TRUE) RETURN;
```

will be entered after every user ministep cycle (except other user AR routines). To initiate tracing, TRAC must be set once.

3.4 I/O Interface

The I/O interface between the MLP-900 and the PDP-10 contains four registers:

- Command/status register — MISC.34
  - DATAO register — MISC.32
  - DATAI register — MISC.33
  - IPL address register — Not addressable

The MLP-900 can read or write these registers as part of the OE miscellaneous-register group; writing these registers is allowed only in microvisor mode. The PDP-10 can read or write these registers via the CONO, CONI, DATAO, and DATAI instructions. The MLP-900 appears to the PDP-10 as two devices on the I/O bus: MLPA, which handles all normal communications, and MLPB, which helps to save and restore the state of the interface.

Table 3.13
Action Requests

Bit	Type	Address	Level	Cause
POWER*	Immediate	7700	ARL.1	Power loss warning
PANIC*	"	7700	"	Interrupt caused by PDP-10
OPAR*	"	7702	ARL.2	Parity error from the odd bank of the Control Memory
EPAR*	"	7704	"	Parity error from the even bank of the Control Memory
SOUF*	"	7706	"	Stack overflow from supervisor mode
SUNF*	"	7710	"	Stack underflow from supervisor mode
UOVF*	"	7712	"	Stack overflow from user mode
UUNF*	"	7714	"	Stack underflow from user mode
CMADR*	"	7716	ARL.3	Control Memory address comparand (MISC.37) matches the Current Address Register while SARM.0 is on
AERR*	"	7720	"	The two adders in the OE differed
BERR*	"	7722	"	Parity error on internal Exchange Bus
PERR*	"	7724	"	Parity error in the translator memory
F.114*	"	7726	"	Two unassigned AR's
F.115*	"	7732	"	
MMERR*	"	7730	"	Main memory parity error
F.117*	"	7734	"	Unassigned

* Indicates a privileged AR.

Table 3.13 (Continued)

Bit	Type	Address	Level	Cause
TASK*	Immediate	7736	ARL.4	Interrupt from the PDP-10
PAGE*	"	7740	"	A CEDE Wait or Store notes that the last translation is bad
SUPVF*	"	7742	"	Attempt by user mode code to execute a privileged ministep or modify a privileged register
PROT*	"	7744	"	An attempt by user mode code to branch into microvisor code at other than an entry point
VADR*	"	7746	"	Virtual address comparand (MISC.37) matches VAR while SARM.1 is on
F.125*	"	7750	"	Three unassigned AR's
F.126*	"	7752	"	
F.127*	"	7754	"	
TRAC	"	7756	ARL.5	Set by user microcode, or by ITRAC after a one-cycle delay
F.131	"	7760	"	Two unassigned AR's
F.132	"	7762	"	
F.133	Wait	7764	"	Five unassigned AR's
F.134	"	7766	"	
F.135	"	7770	"	
F.136	"	7772	"	
F.137	"	7774	"	

3.4.1 Command/Status Register

The command/status register (MISC.34) is a 27-bit register, as shown in Figure 3.24.

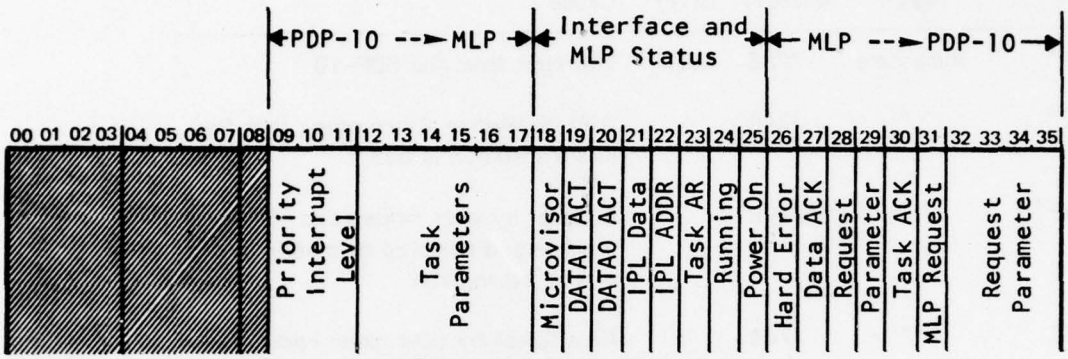

Figure 3.24 Command/Status Register Format

Bits	Use
9-11	Priority-interrupt level
12-17	Task parameter (provided by the PDP-10 along with a TASK AR)
18	Microvisor mode
19	DATAI-active
20	DATAO-active
21	IPL-data mode
22	IPL-address mode
23	TASK-AR pending (F.120)
24	MLP-running (F.164)
25	MLP Power-Up
26	Hard-error PI (priority interrupt)
27	Data-ack PI
28,29	Request parameter (expanding on the MLP-request PI)
30	Task-ack PI
31	MLP-request PI
32-35	Request parameter (expanding on the MLP-request PI)

3.4.2 DATAO and DATAI

DATAO (MISC.32) and DATAI (MISC.33) are 36-bit data-transmission registers, usable in either direction. Each is accompanied by an "active" bit in the command/status register. Writing into one of these registers by either the PDP-10 or the MLP-900 sets the register's active bit; reading it clears the active bit (without altering the data). Note that an MLP-900 user may read these registers (and, by so doing, clear the active bit).

3.4.3 MLP-900 Interface Manipulation

The MLP-900 can read the command/status register and the DATAO and DATAI registers via a GENT ministep. In addition, if SUPVLB (F.176) is true, the following command/status fields are available directly as pseudo-flops and pointers:

Field	Found In
Task parameter (bits 12-17)	P.17
DATAI-active (bit 19)	F.326
DATAO-active (bit 20)	F.327
Hard-error PI (bit 26)	F.320
Data-ack PI (bit 27)	F.321
Task-ack PI (bit 30)	F.322
MLP-request PI (bit 31)	F.323

In microvisor state the MLP-900 can load the Command/Status, DATAO, or DATAI registers via a GENT ministep. Writing the command/status register loads only bits 26-35; bits 0-25 cannot be written directly. Furthermore, if the MLP request PI (bit 31) is zero ("new value"), the MLP-900 request parameter (bits 28, 29, and 32-35) is ignored and that field of the command/status word is cleared. Setting one or more of the four PI bits (26, 27, 30, or 31) causes the MLP-900 to interrupt the PDP-10 on the priority interrupt level specified by bits 9-11 (if the interrupt level is not zero); while their names suggest distinct functions, the four PI bits perform identically.

3.4.4 PDP-10 Interface Manipulation

The PDP-10 recognizes the MLP-900 as two devices on the I/O Bus: MPLA is device 424 and MLPB device 434, both octal. The PDP-10 DATAI and DATAO operations on these devices transfer 36-bit values to and from the DATAI and DATAO registers; the active bits are set by a DATAO operation and reset by a DATAI operation. On device MLPA, the DATAO operation loads DATAO and the DATAI operation reads DATAI. On device MLPB, however, the DATAO operation loads DATAI and the DATAI operation reads DATAO. The PDP-10 CONI and CONO operations transfer 18 bits to and from the command/status register, respectively:

CONO, MLPA; Commands Out

Bits	Function
18-20	New priority interrupt level
21	Set IPL mode
22	Set panic AR (F.101)
23	Set task request (F.120)
24	Set/reset clock (F.164)
25	Reset interface
26	Reset hard error PI
27	Reset data ack PI
28	Reset task ack PI
29	Reset MLP request PI
30-35	New task parameter

CONI, MLPA; Status In

Bits	Reads
18-25	Bits 18-25 of command/status register
26-29	Bits 26, 27, 30, and 31 of command/status, the four PI bits.
30-35	Bits 28, 29, 32-35 of command/status register, the MLP-900 request parameter

CONO, MLPB; a NOP

CONI, MLPB; Read Commands

Bits	Reads
18-20	Priority Interrupt Level
21,22	Zero
23,24	Bits 23, 24 of command/status
25-29	Zero
30-35	Bits 12-17 of command/status (PDP-10 task parameter)

In general, MLPB is needed only to save the state of the interface; all normal communication is done via MLPA.

3.4.4 IPL Mode

IPL mode is used to load MLP-900 control memory directly over the I/O bus. IPL mode is initiated by a CONO to MLPA that sets IPL-mode (bit 21). This puts the MLP-900 into IPL-address mode; the next DATAO to MLPA loads the IPL-address register and puts the MLP-900 into IPL-data mode. Subsequent DATAO's to MLPA are used to load successive control memory locations, with the control-memory mode bit set to 2 (supervisor mode); the IPL-address register is incremented *prior* to each control memory store. IPL-mode is terminated by any CONO to MLPA.

Chapter 4

General Purpose Microprogramming Language

The General Purpose Microprogramming (GPM) language is an implementation language developed by the PRIM project as a machine-dependent microprogramming language for the MLP-900. It is essentially a generalization of the machine-level language forms presented in Chapter 3. Its design follows an assembly language philosophy, which allows the programmer to predict all instruction sequences and requires no run-time support system, although syntactic block structure and high-level control structures are provided to assist the programmer. GPM is the primary language for the MLP-900 (no assembly language exists) and, as such, was designed to be used for both diagnostics and emulators.

The syntax of GPM is given in this and the previous chapter as modified BNF definitions. Each definition is preceded by a definition number within braces; each reference to that definition is immediately followed by its definition number within braces so as to facilitate cross references. All syntax equations before program{63} are in Chapter 3; the remainder are in this chapter. The few primitive constructs referenced in definitions are given in italics, as in *emptystring*.

4.1 Program Structure

A GPM program starts with a title declaration; the title identifier must be nonreserved (see Appendix C). The body of the program has two parts: a declaration list and statement list.

Syntax:

{63} program ::=
TITLE id{1} body{64} closing{70}

{64} body ::=
declarationlist{65} ; stmtlist{66} | stmtlist{66}

{65} declarationlist ::=
declaration{67} | declarationlist{65} ; declaration{67}

{66} stmtlist ::=
statement{69} | stmtlist{66} ; statement{69}

4.1.1 Declarations

Declarations define conditions that will be active for the scope of the body in which they are made.

Syntax:

{67} declaration ::=
pseudodeclrtn{72} | TEMPORARY rlist{68} |
EQUATE id{1} id{1} | EQUATE id{1} id{1} number{5}

{68} rlist ::=
R..37 | rlist{68} *R..37*

4.1.1.1 EQUATE Declaration

There are two forms of the EQUATE statement. The first takes two symbols and equates the first to the second (*i.e.*, the first will be treated as if it were the second). For example, after the declaration EQUATE PC R.3; every occurrence of PC within the scope of the declaration will be interpreted as R.3. The following examples are legal EQUATE statements of this form:

```
EQUATE INDEX P.6;
EQUATE MINUS.ONE 777777777777;
EQUATE EQ EQUATE;
EQ INFINITE.LOOP.START DO.BEGIN;
```

The second EQUATE form takes two indexed identifiers and a number as arguments; it is used to equate blocks of indexed names. For example, after the declaration

```
EQUATE AC.0 R.10 10 ;
```

every occurrence of AC.0 through AC.7 within the scope of the declaration will be interpreted as R.10 through R.17, respectively.

4.1.1.2 TEMPORARY Declaration

The TEMPORARY declaration declares general registers that may be used as temporaries by the GPM code generators. This declaration allows more complicated arithmetic operations and data transfers to be compiled.

4.1.2 Statements

The statement types are discussed in detail in Section 4.2. All statements may be tagged by one or more identifiers, which can be used as statement labels. Reserved identifiers, numbers, and nonalphanumeric characters may not be used as statement labels.

Syntax:

{69} statement ::=
id{1} : statement{69} | substmnt{71}

4.1.3 Closing

A GPM program is closed with the reserved word FINISH, optionally followed by an identifier. This identifier, if present, specifies the starting statement label of the program to the MLP loader.

Syntax:

{70} closing ::=
FINISH | FINISH id{1}

4.2 Statement Types

Six classes of statement may appear in GPM programs: pseudodeclarations, assignment statements, control statements, switch tags, low-level statements, and constants. Pseudodeclarations, which are discussed in Section 4.3, do not generate any code and only condition the compilation or listing generation that follows. Assignment statements, which are discussed in Section 4.4, evaluate expressions and move data within the MLP-900. Control statements, which are discussed in Section 4.5, determine the control flow of the program. Switch tags identify entry points into switch-selected code sequences; they are discussed in Section 4.5.6. Low-level statements each compile to exactly one ministep; they, and constants, are presented in Section 4.6.

Syntax:

{71} substmnt ::=
pseudodeclrtn{72} | assignment{76} | control{103} | switchtag{114} |
lowlevel{117} | constant{118}

4.3 Pseudodeclarations

Four types of pseudodeclaration may appear anywhere in a GPM program: ORIGIN statement, COMMENT statement, INCLUDE statement, and output-control statements. The pseudodeclarations ignore block boundaries.

Syntax:

{72} pseudodeclrtn ::=
ORIGIN number{5} ; | COMMENT *any-string-not-containing-a-semicolon* ; |
outputctrl{73} ;

{73} outputctrl ::=
PRINTOFF | PRINTON | outputtype{74} modeset{75}

{74} outputtype ::=
HEXADECIMAL.CODE | NORMAL.CODE | LABEL.TABLE

{75} modeset ::=
MODE TRUE | MODE FALSE

4.3.1 ORIGIN

The GPM compiler produces absolute code. The ORIGIN statement is provided to allow the programmer to specify where the code that follows should be placed in control memory. The number in the origin statement is the location to receive the next instructions compiled. All succeeding instructions will be placed in successive locations. The initial value for the origin is 0.

4.3.2 COMMENT

The COMMENT statement is provided to allow the programmer to document his program. In addition to the COMMENT statement, there is also a feature to allow a comment to be entered on each line as one might do in assembly code. This feature causes any string starting with an exclamation point (!) and continuing through the following end-of-line to be interpreted by the compiler as a semicolon.

Example:

```
COMMENT comment facility example;
R.0 ← 0                 !zero general register zero
R.1 ← R.0 + 1 !         set general register one to one
COMMENT                 end of comment facility example !!!!!!!
```

4.3.3 INCLUDE

The INCLUDE feature may be used anywhere within a GPM program. It is simply "INCLUDE" followed by a standard TENEX file name. Included files may INCLUDE other files. It is good practice when working with INCLUDE files to use the proper directory name within the file, so the file can be used by others.

Example:

```
PRINTOFF
COMMENT sample include file ;
BEGIN NAMED INCLUDE.FILE.SAMPLE
EQUATE R.5 INPUT !setup some register definitions
EQUATE R.13 OUTPUT ;
INCLUDE <OESTREICHER>SQUARE-ROOT.INC
COMMENT if this is used when not connected to <OESTREICHER> it will still work ;
END NAMED INCLUDE.FILE.SAMPLE !close any open blocks
PRINTON
```

4.3.4 Output Control

Several pseudodeclarations are provided to control the generation of the source listing and the code listing. A complete listing consists of four parts:

- The reformatted source file with errors flagged and corrected (where possible)
- The label table
- The compiled code listed in octal (normal code)
- The compiled code listed in hexadecimal

4.3.4.1 Source Listing Control

Two pseudodeclarations control the generation of the source listing: PRINTOFF and PRINTON (see the example in Section 4.3.3). PRINTOFF will always turn off the listing; PRINTON will turn on the listing only if there has been one PRINTON for each preceding PRINTOFF, thus enabling the user to nest PRINTOFF-PRINTON pairs. This is useful with nested INCLUDE files (see Section 4.3.3), which usually are not desired in the output listing. There is a compiler switch to allow all PRINTOFF's to be ignored, thus forcing a complete listing (see Section 4.7).

4.3.4.2 Code Listing Control

A pseudodeclaration exists to control each of the three other parts of the output listing. If several of these statements appear, the last one will be in effect when the listings are generated at the end of the compilation. The initial settings are

```
LABEL.TABLE MODE FALSE;
NORMAL.CODE MODE FALSE;
HEXADECIMAL.CODE MODE FALSE;
```

Compiler switches exist to change these initial settings (see Section 4.7).

4.4 Assignment Statements

The five types of assignment statements are

- Arithmetic. Assigns the value of an arithmetic expression to a general register (OE).
- Boolean. Assigns the value of a boolean expression to a flop (CE).
- Data Transfer. Copies data from one machine register to another (OE and CE).
- Increment or Decrement. Increments or decrements a pointer's value.
- Shift. Shifts a general register's contents and replaces them.

Syntax:

```
{76}  assignment ::=
          arithmetic{77} | boolean{84} | datatransfer{89} | incrdecr{100} | shreg{101}
```

4.4.1 Arithmetic Assignments

The arithmetic assignment statement has three parts: assignment to a result register, an arithmetic expression, and modifiers. Only the arithmetic expression must be present. The first two parts define an ordinary arithmetic calculation, while the modifiers condition the evaluation of the expression.

Syntax:

```
{77}  arithmetic ::=
          aa{8} ← arithmetic{77} ; | aexp{78} amod{82} ;

{78}  aexp ::=
          aterm{79} | aterm{79} aop{81} aexp{78}
```

{79} aterm ::=
aprimary{80} | NOT aprimary{80}

{80} aprimary ::=
aa{8} | number{5} | *P..7* | (arithmetic{77})

{81} aop ::=
+ | - | PLUS | MINUS | AND | OR | XOR

{82} amod ::=
amask{12} testmode{13} ashift{83} | ashift{83} amask{12} testmode{13} | . . .
<<amask, testmode, *and* ashift *may be specified in any order*>>

{83} ashift ::=
shdir{15} number{5} | *emptystring*

4.4.1.1 Mask (amask)

If no mask modifier is specified, "(M.0)" is used. In nested expressions, the outer specification (if there is one) will replace the default value. The mask "*(M..17)*" or "*[M..17]*" specifies which mask register will be used for the calculation; parentheses around the mask register indicate that clear mode is false and brackets indicate that clear mode is true.

4.4.1.2 Test Mode (testmode)

Test mode is set if the test-mode symbol (*) is present, but the preferred method of specifying test mode is by omitting the assignment (see Section 4.4.1.5). For nested expressions, each test-mode symbol complements the test-mode bit.

4.4.1.3 Shift (ashift)

If no shift is specified, none will occur. Right shift (divide) is specified by a "/" and left shift (multiply) is specified by a "\." Extra ministeps will be generated if the shift amount is not one given in Table 3.3.

4.4.1.4 Operators (aop)

The unary one's-complement NOT is of highest precedence. No precedence is associated with any of the binary operators (aop). If order of evaluation is important, it must be controlled with parentheses. The binary operators are

+	Two's complement add
-	Two's complement subtract
PLUS	Long add (see Section 3.2.2.1)
MINUS	Long subtract (see Section 3.2.2.1)
AND	Logical *and*
OR	Logical *or*
XOR	Logical *exclusive or*

4.4.1.5 Result (aa ←)

If no assignment of a result is specified, the operation will be done with test mode true. The result register can be specified directly, or indirectly through a pointer register. Both * *P..7* and @ *P..7* specify indirect references to the general registers. The character @ is a normal indirect; the register number is taken from the five low-order bits of the specified pointer register. The character * is a special indirect; it acts like a normal indirect, except that the low-order bit is forced to 1 in the register number.

Examples:

```
COMMENT if R.4 = R.11 GOTO equal.tag ;
R.4 XOR R.11 !result will be zero on equals
IF ZSP GOTO EQUAL.TAG ;

COMMENT M.1 contains 7700, M.2 contains 77770 ;
COMMENT number in R.3 field M.1 added to R.4 field M.2 ;
R.4 ← R.4 + ( R.3 [M.1] /3 ) (M.2) ;
```

4.4.2 Boolean Assignments

The boolean assignment statement provides a method to set flops to the value of a boolean expression. The boolean expression is composed of flop names, the boolean constants TRUE and FALSE, and the logical operators AND, OR, XOR, and NOT. As in the arithmetic expression, the unary one's complement NOT is of highest precedence and there is no precedence among the binary operators (bop). If order of evaluation is important, it must be specified with parentheses. The boolean assignment is a generalization of the MAST ministep, which is limited to expressions involving at most two flops; no temporary storage is used by the boolean assignment in evaluating more complex expressions.

Examples:

```
F.3 ← F.3 XOR F.5 !if F.5 then complement F.3
F.7 ← F.1 OR F.2 OR NOT F.3 ;
F.11 ← (F.0 AND F.5) OR NOT (F.7 AND F.6);
```

Syntax:

```
{84}  boolean ::=
          F..277 ← bexp{85} ;

{85}  bexp ::=
          bexpr{86} | boolean{84}

{86}  bexpr ::=
          bterm{87} | bexp{85} bop{43} bterm{87}

{87}  bterm ::=
          bprimary{88} | NOT bprimary{88}

{88}  bprimary ::=
          F..377 | TRUE | FALSE | ( bexp{85} )
```

4.4.3 Data Transfers

The left and right sides of the data transfer statement must represent data objects of matching size. The possible sizes are 36, 16, and 8 bits. The optional ***dtnot*** in the 8-bit transfer causes a one's complement NOT. Left and right operands may not both be OE registers for 8-bit or 16-bit transfers.

Syntax:

{89} datatransfer ::=
dxfr36bits{90} | dxfr16bits{92} | dxfr8bits{98}

{90} dxfr36bits ::=
oeloc{25} ← dt36source{91} ;

{91} dt36source ::=
oeloc{25} | number{5} | *P..7*

{92} dxfr16bits ::=
oeloc{25} *H..1* ← ceregpair{93} ; |
ceregpair{93} ← dtnot{95} dt16source{96} cemask{97} ;

{93} ceregpair ::=
(cereg{94}) | (cereg{94} , cereg{94}) | *S..17*

{94} cereg ::=
CE..137 | *P..7* | *XBUS..3*

{95} dtnot ::=
NOT | *emptystring*

{96} dt16source ::=
oeloc{25} *H..1* | ceregpair{93} | number{5}

{97} cemask ::=
(number{5}) | [number{5}] | *emptystring*

{98} dxfr8bits ::=
oeloc{25} *B..3* ← dtnot{95} cereg{94} ; |
cereg{94} ← dtnot{95} dt8source{99} cemask{97} ;

{99} dt8source ::=
cereg{94} | oeloc{25} *B..3* | number{5} | *F..377*

The mask notation is similiar to that in arithmetic assignment, except that the mask is specified as an immediate constant instead of as a mask register. The parentheses specify a normal mask, where all masked-out bits (zero mask bits) remained unchanged. The brackets specify a clear mask, where all masked-out bits are zeroed. If no mask is specified, an all-ones mask of the appropriate size is used. Transfers to the OE cannot be masked!

4.4.3.1 36-bit Transfers

The 36-bit left operands are OE registers. The right operands are OE registers, constants, or pointer registers. In the case of pointer registers, the high-order 28 bits are zero. A 36-bit transfer generates either one or two GENT ministeps; transfers that cannot be done in a single ministep (*e.g., M..17 ← constant*) require a TEMPORARY register for the intermediate result. The OE registers are

- ***R..37*** 32 general-purpose registers
- ***M..17*** 16 mask registers
- ***MISC..37*** 32 miscellaneous registers
- ***A..1777*** 1024 auxiliary-memory registers
- ***XLATOR..777*** 512 translator-memory registers (only microvisor-mode access allowed)

OE registers may be referenced directly, or indirectly through a pointer register. OE registers are divided into pages of up to 256 registers. The 8-bit pointer registers can address any register within a page. It is possible to address registers indirectly only within single designated pages. As with the arithmetic assignment statement, the * indirect operator will force the low-order register number bit to a 1.

4.4.3.2 16-bit Transfers

There are two types of 16-bit left operands. A 16-bit transfer in which an OE location is the destination is limited to a single GENT-MOVE pair of ministeps; a transfer from the OE, or entirely within the CE, generates one or more MOVE ministeps. These and constants comprise the possible right operands. The two left operand types are

(1) OE register half-words -- oeloc *H..1*
Half-words are numbered from left to right. The high-order four bits are not referenced. Thus H.1 refers to the low-order 16 bits and H.0 refers to the next lowest 16 bits. Note that whenever half-word references are used as the left side of a data transfer, the remainder of the specified OE register is zeroed. Note additionally that OE registers may not appear as both left and right operands.

(2) CE register pair -- (cereg) or (cereg, cereg) or *S..17*
The CE register-pair construct references an even/odd pair of CE registers. If only a single CE register is named within the parentheses, the designated register is treated as if it were the first of an explicitly named pair and the other register from the even/odd pair is taken as the second. The two examples following will each cause a swapped data transfer; the first will transfer (P.1, P.0) into R.0 and the second will transfer (P.4, P.5) into (P.1, P.0):

Examples:

```
R.0 H.1 ← (P.1);
(P.1) ← (P.4);
```

If both CE registers are named explicitly within the parentheses, they must be in the same even/odd pair; otherwise, they cannot be moved to or from an OE register half-word. The following is an impossible data transfer because P.1

and P.2 are not both in the same even/odd CE register pair; the transfer occurs as if (P.1, P.0) had been specified:

(P.1, P.2) ← R.17 H.0;

The construct S.*n* is equivalent to (CE.100+2*n*) or (CE.100+2*n*, CE.101+2*n*).

4.4.3.3 8-bit Transfers

An 8-bit transfer in which an OE location is used as either source or destination generates a GENT-MOVE pair of ministeps; a transfer entirely within the CE generates one or more MOVE ministeps. There are two types of 8-bit left operands:

(1) OE register byte -- oeloc *B..3*
Bytes are numbered from left to right. The high-order four bits are not referenced. Therefore B.3 refers to the low-order 8 bits, B.2 refers to the next lowest 8 bits, *etc.* Note that if the OE register is a left operand, masking is ineffective since it is performed in the CE as the store takes place; also, the bytes of the OE register that were not specified are zeroed. Note additionally that OE registers may not appear as both left and right operands.

(2) CE register -- cereg
The CE registers are:

- *CE..137* all CE registers
- *P..7* pointer registers (CE.40-CE.57)
- *XBUS..3* CE exchange bus (CE.70-73 as left operands; CE.64-67 as right operands)

In addition to the two operand types discussed above, 8-bit right operands may also be either constants or flops. In the case of flops, the right operand is interpreted as an 8-bit quantity where all bits contain a copy of the value of the specified flop.

Examples:

```
R.0 ← NOT A.173 [777];
A.PG.0 @ P.1 ← A.PG1 @ P.1;
M.17 H.1 ← NOT S.12;
M.1 ← 777777777777;
R.3 B.3 ← P.17;
R.3 ← P.17;
P.17 ← CE.0;
P.3 ← NOT F.144 (123);
```

4.4.4 INCREMENT and DECREMENT

An increment or decrement statement allows a constant to be added to or subtracted from a pointer register, respectively. When one of these statements is followed by an unlabeled conditional branch, the compiler may generate a BRAD that incorporates (part of) both statements.

Syntax:

{100} incrdecr ::=
bradop{49} *P..7* BY number{5}

4.4.5 SHIFT

The shift instructions provide for single- and double-register shifting by fixed or variable amounts. One or more SHIN ministeps are generated, depending on the shift amount.

Syntax:

{101} shreg ::=
shop{30} aa{8} shdir{15} shamt{102} shmask{32} testmode{13} ;

{102} shamt ::=
@ | number{5}

4.5 Control Statements

There are six types of control structures in GPM:

- Blocks Prototype compound statement form (see Section 4.5.1)
- BREAK Standard block exit mechanism (see Section 4.5.2)
- Branch Unconditional transfer of control (see Section 4.5.3)
- DO Looping mechanism (see Section 4.5.4)
- IF Conditional execution and compilation (see Section 4.5.5)
- Switch Case analysis (index branch) mechanism (see Section 4.5.6)

Syntax:

{103} control ::=
block{104} | break{106} | branch{107} | do{110} | if{111} | switch{112}

4.5.1 Blocks

The BEGIN...END block is the prototype compound statement form in GPM. The DO.BEGIN (see Section 4.5.4), IF...THEN.BEGIN (see Section 4.5.5), and SWITCHON...INTO.BEGIN (see Section 4.5.6) statements are special cases of blocks. All have the characteristics of the standard block in addition to special features of their own.

Syntax:

{104} block ::=
BEGIN name{105} body{64} END name{105} ;
<<both instances of name *must be identical>>*

{105} name ::=
NAMED id{1} | *emptystring*

The block defines the scope for any declaration statements that may appear in the block body. In the special blocks, the BEGIN...END also delimits the scope of the control structure involved. Blocks can be named by following the BEGIN with "NAMED *name*," which enables the program to reference the block by name. This is used for two purposes: first, an END may be named, thus closing all blocks within the named block; second, the block name may be used by the BREAK statement to specify which block to exit.

4.5.2 BREAK

The BREAK statement will cause program control to branch to the end of the named block, except that if no name is supplied with BREAK, the current block will be exited. Note that this is different from a RETURN statement: RETURN exits a subroutine to the called location (determined at runtime), whereas BREAK exits a block to its end (determined at compile time).

Syntax:

{106} break ::=
BREAK *name*{105} ;

4.5.3 Branches

There are three types of unconditional branches: CALL, RETURN, and GOTO. The CALL statement pushes the location of the next sequential instruction in control memory onto the top of the hardware subroutine stack and goes to that location. The RETURN statement transfers control to the location on the top of the subroutine stack and pops the stack. The GOTO simply branches to the specified as the branch destination. In addition to the unconditional branches provided by the branch statements, GPM also has conditional branches; these are special forms of the IF statement described in Section 4.5.5.

Syntax:

{107} branch ::=
CALL bnchdest{108} ; | RETURN ; | GOTO bnchdest{108} ;

{108} bnchdest ::=
location{109} | < *P..7* > | location{109} < *P..7* >

{109} location ::=
trfrlabel{56} | offset{45} | id{1} offset{45}

The form <*P..7*> in a branch destination represents an offset, either from the continuation address (the next instruction word) or from any location that is supplied immediately preceding it.

There are two types of branch-destinations: relative (offset) and absolute. Either type can be indexed by the value of a pointer register. With indexing, the unindexed branch location is always calculated first and the value of the pointer register is then added. This addition might cause overflow, in which case the branch destination will wrap around to low control memory. If the branch destination is only a pointer register (no location supplied), then the index is relative to the next sequential instruction in control memory. An absolute destination may reference a statement-label identifier (see Section 4.1.2) or an absolute

location specified by a number. A relative destination may be merely an offset relative to the current location in control memory or an offset from some specified statement-label identifier.

Example:

```
        GOTO TAG;
        CALL 100 <P.3>;
    TAG:
        CALL TAG +3;
        RETURN
        GOTO -4;
        CALL +1<P.1>;
```

4.5.4 Loops

The DO.BEGIN...END statement unconditionally repeats the body of code contained within the block. This is the looping construct in GPM. The loop must be exited with a BREAK command.

Syntax:

{110} do ::=

```
        DO.BEGIN name{105} body{64} END name{105}
        <<both instances of name must be identical>>
```

4.5.5 Conditional Control

There are two types of conditional-control statements: block-structured and conditional-branch. The first is for the conditional execution of sections of code and the second for the conditional transfer of control. The first is sufficient in all cases, but the second is easier and more efficient where appropriate. Note that the form "THEN.BEGIN name ... END name" is a special form of a block (see Section 4.5.1).

Syntax:

{111} if ::=

```
        IF bexp{85} THEN.BEGIN name{105} body{64} ELSE stmtlist{66} END name{105} ; |
        IF bexp{85} THEN.BEGIN name{105} body{64} END name{105} ; |
        IF bexp{85} BREAK name{105} ; | IF bexp{85} RETURN ; |
        IF bexp{85} CALL id{1} ; | IF bexp{85} GOTO id{1} ;
        <<both instances of name must be identical in each of the first two forms>>
```

4.5.5.1 Block-structured IF Statement

The block-structured IF statement has two forms, the most general of which is the "IF *boolean-expression* THEN.BEGIN...ELSE...END" form. If the boolean expression is true, the body following the THEN.BEGIN will be executed and the statement list following the ELSE will not be executed. If the boolean expression is false, the opposite will happen: the body will not be executed and the statement list will be. Any declarations that follow the THEN.BEGIN will be active both for statements in the body following the THEN.BEGIN and for the statement list following the ELSE. The second form of IF simply omits the ELSE sections.

Each boolean expression is evaluated at compile time. If it evaluates to the constant TRUE or FALSE in an IF statement, then only code for the appropriate statements will be compiled and no test will be compiled at all. ORIGIN's and statement-label assignments can also be conditionally compiled using this facility. There is no way to specify declarations conditionally for a block.

4.5.5.2 Conditional-branch IF Statement

The conditional-branch IF statement does not contain either the THEN.BEGIN or the END of the block-structured IF statement. Immediately following the boolean expression is a branch statement (BREAK, CALL, RETURN, or GOTO). The branch statements are restricted, however, in that only statement-label names may be used for the CALL or GOTO destinations. Note that a BREAK inside a block-structured IF statement will exit only that IF-block if the BREAK is not NAMED. This means that the following two statements are *not* equivalent:

```
IF ZSP THEN.BEGIN BREAK END;
IF ZSP BREAK;
```

4.5.6 Switches

A switch statement generates a control structure consisting of an indexed branch into a switch table that follows the code generated by statements within a switch block (which, in turn, branches around the switch table). The switch table contains branches to code generated for statements following switch tags that occurred within the switch block. The switch table has one entry for each possible index value from zero through the largest switch value declared in a switch tag (within that switch block).

Syntax:

{112} switch ::=
SWITCHON < *P..7* > switchblk{113} ;

{113} switchblk ::=
INTO.BEGIN name{105} body{64} END name{105}
<<both instances of name *must be identical>>*

4.5.6.1 Switch Tags

There are two switch-tag statements: ENTRY and CASE. The ENTRY statement specifies a list of pointer-register values that are to cause control to transfer to the first statement following the ENTRY statement. The CASE statement is equivalent to the ENTRY statement except that an initial BREAK out of the switch block precedes the entry point to prevent execution of a prior ENTRY or CASE from dropping into the statements associated with the current CASE.

Syntax:

{114} switchtag ::=
ENTRY switchlist{115} ; | CASE switchlist{115} ;

{115} switchlist ::=
switchvalue{116} | switchlist{115} , switchvalue{116}

{116} switchvalue ::=
number{5} | number{5} THRU number{5} | number{5} THRU |
THRU number{5} | THRU

4.5.6.2 Switch Values

Switch values are either numbers or ranges of numbers. The maximum range of a switch value is 0 through 377, octal. On the THRU version of the switch value, 0 is assumed for an unspecified start and 377 is assumed for an unspecified end. Also, if some particular number has been assigned previously, the THRU specification will ignore it. On the other hand, a single number specification will override.

4.5.6.3 Programming Considerations

Each switch value declared produces one instruction of overhead. The switch is assumed to have a zero origin. For example, "CASE 2,4" will have five (0-4) instructions of overhead. No run-time check is made on the value of the pointer register. Any unspecified values below the maximum specified value will transfer control to the location immediately following the switch table. Values above the maximum, however, will transfer to a location beyond the switch table, producing unexpected results. The first executable statement following the INTO.BEGIN of a switch block (other than declarations) should be an ENTRY statement; a CASE will produce an unnecessary BREAK.

Example:

```
SWITCHON <P.1> INTO.BEGIN
   ENTRY 2,4;
            COMMENT CASES 2,4;
   CASE 1 THRU 6,10;
            COMMENT CASES 1,3,6,10;
   ENTRY 5;
            COMMENT CASES 1,3,5,6,10;
END
```

4.6 Low-level and Constant Statements

The low-level GPM statements are

- GEAR (see Section 3.2.2.1)
- CEDE (see Section 3.2.2.2)
- SHIN (see Section 3.2.2.3)
- GENT (see Section 3.2.2.4)
- BRAT (see Section 3.3.2.1)
- BENT (see Section 3.3.2.2)
- BORE (see Section 3.3.2.3)
- BRAD (see Section 3.3.2.4)
- BEAD (see Section 3.3.2.5)

- BLOT (see Section 3.3.2.6)
- MAST (see Section 3.3.2.7)
- MOVE (see Section 3.3.2.8)

Syntax:

{117} lowlevel ::=
gear{7} | cede{19} | shin{29} | gent{34} | brat{40} | bent{46} |
bore{47} | brad{48} | bead{50} | blot{57} | mast{59} | move{60}

A constant statement generates one word containing the 32 low-order bits of the number.

Syntax:

{118} constant ::=
number{5}

4.7 The GPM Compiler

The GPM compiler is available as a TENEX subsystem under the name GPM. The GPM command prompt is a double colon; a command consists of a single letter and is executed immediately. The "C" (compile) command prompts for its source, binary, and listing files. Compilation begins as soon as the last file is confirmed.

Example:

```
@GPM

MLP-900 Language System
    Type ? for help
MONDAY, NOVEMBER 11, 1974 14:29:01-PST
USED  0: 0: 0. 5 IN  0: 0: 1.45
Compiler Version GPM.4.74.7
::H  HEXADECIMAL.CODE MODE TRUE
::L  LABEL.TABLE MODE TRUE
::C
source file:PROGRAM.GPM;6 [Old version]
binary file:PROGRAM.BIN;6 [Old version]
listing file:PROGRAM.LST;1 [New version]

↑L
%PROGRAM.NAME        GPM.4.74.7        11-NOV-74 14:30:57 Pg 20 %
        %**No Errors Detected**%

::Q
MONDAY, NOVEMBER 11, 1974 14:31:02-PST
USED  0: 0: 20.20   0: 2: 2.30
```

If no binary file is desired, a null binary file (*NIL:*) should be specified. The same is true for the listing file (the compilation will run more quickly if no listing is generated).

The listing file can be recompiled without any editing. One should be careful, however, since the compiler will "correct" all errors in the source file, and these "corrections" will disappear after recompiling the listing file.

The set of GPM commands is

C Compile. Compiles GPM source program (shown in above example).
F Fast compilation. Sets flag for syntax check; no code generation.
H HEXADECIMAL.CODE MODE.[9]
L LABEL.TABLE MODE.[9]
N NORMAL.CODE MODE.[9]
P PRINTON. Forces complete listing; sets flag to suppress any PRINTOFF statements in the program source.
Q Quit.
S Switch status. Prints the current switch settings as determined by the commands F, H, L, N, and P.
T Teletype Test-compile. Same as C, except that binary file is *NIL:* and both source and listing file are *TTY:*

A complete GPM listing contains four parts:

- The source programs with errors flagged and corrected (where possible).
- The label table.
- The compiled code listed in octal (normal code).
- The compiled code listed in hexadecimal.

Section 4.3 discussed the GPM pseudostatements that affect whether or not these listings are produced. This section discusses in detail the contents of each part of the listing.

4.7.1 Source Program

The source listing is primarily a reformatted copy of the input with a few changes. The most important change is that all text bracketed by percent (%) delimiters is lost, along with the delimiters, because the compiler itself uses % in the listing file to delimit page headings and error messages, which are not proper parts of the listed program.

The output of the GPM compiler can be fed back into the compiler and processed, usually with fewer errors. In attempting to correct errors, the compiler either inserts what it believes to be a missing symbol or "erases" offending symbols by enclosing them in % delimiters. If all the corrections made in the output listing (possible new source file) are satisfactory, no recompilation is necessary.

4.7.2 Label Table

The label table is output after the FINISH statement and is bracketed by percent-signs. It has three columns: octal location, hexadecimal location, and label name.

---

9. This command controls the generation of a section of the GPM listing. The control setting alternates every time the command is entered; the new value is printed by the GPM compiler. The initial value is false (i.e. no output).

Example:

```
% LABEL TABLE      %
% 7702    FC2      TAGA    %
% 7750    FE8      TAGB    %
```

4.7.3 Code Listings

The code listing comes in five fields. The first three are the location of the code word, a flag digit, and the op code. The fourth field contains the instruction coding; the fifth field is a translation of the single MLP instruction back into a GPM statement. This last field is provided for easier reading of the compiled code.

The flag digit is not copied to the MLP by the loader. The 1 flag marks long immediate instructions and causes the location-counter value to advance two instead of one. The 4 and 2 flags mark ORIGIN's and labels.

In a normal (octal) listing, the location and instruction-code fields are in octal.

Example:

```
%7701 0 BEAD 2 121 7027  /IF TRUE THEN GOTO 7027 END;%
%7702 1 GEAR 4   0 37 77  R.37 ←R.37 OR NOT 777777777657(M.0);%
%7704 0 GENT 0   2 33 36  MISC.33 ←R.36;%
```

A hexadecimal listing is the same as the normal one, except that the location and instruction fields appear in hexadecimal instead of octal.

Example:

```
%FC1 0 BEAD 2 91   E17  /IF TRUE THEN GOTO 7027 END;%
%FC2 1 GEAR 4  0 1F CF  R.37 ←R.37 OR NOT 777777777657 (M.0);%
%FC4 0 GENT 0  2 1B CB  MISC.33 ←R.36;%
```

Appendix A
Additional Exec and Debugger Commands

The general PRIM exec and debugger commands are discussed fully in ***PRIM System: User Reference Manual***, which the emulator writer is expected to have read. This appendix discusses exec and debugger commands or subcommands not in the reference manual. PRIM keeps a flag, known as the "whiz" flag, that gives the user access to additional facilities and commands not required by the emulator user. When a user runs PRIM directly with the command

@<PRIM>PRIM

he begins as a whiz; when he gets to PRIM indirectly by running a working emulator, he is not a whiz. An additional intervention character is available to the whiz during emulator execution:

> ***MLP-halt*** (initially cntl-Q) halts the emulator at an arbitrary point between MLP-900 cycles. This halt does not require the cooperation of the emulator as does the ***abort*** intervention (which sets the QUIT AR bit, F.132).

A.1 Exec Commands

The following commands are either not discussed in the reference manual in their entirety or have privileged subcommands that are not discussed:

CHANGE
ENABLE
LOAD
NO
SAVE
TABLES

There are several additional, undocumented, privileged commands specifically for the PRIM developers that should be ignored by other privileged users. In particular, any command involving the name "6-12" (which is the name of the debugger for the PRIM framework itself) should be avoided.

Change additionally allows the MLP-halt intervention character to be changed.

Enable sets flags that control the state of the PRIM framework. For the nonprivileged user, only the whiz state may be enabled. When whiz has been enabled, all the features discussed in this appendix are available. CALL-STOP and STOP-STOP are particularly useful in the earliest stages of emulator debugging, since jointly they disable all PRIM framework servicing of the emulator. IO-TRACE and RESUME-STOP are more useful when the emulator basically works.

```
>ESCNABLE ? WHIZ
ENABLE wESCHIZ CR
>ESCNABLE ?  One of the following:
 CALL-STOP
 IO-TRACE
 RESUME-STOP
 STOP-STOP
 WHIZ
 XOFF
ENABLE xESCOFF CR
>
```

CALL-STOP causes the PRIM framework to print--rather than process--the value of the call parameter contained in R.37 on an emulator call to MLP.CALL and to stop the emulator. Because the microvisor returns control to the emulator as soon as the call parameter has been passed to the TENEX MLP driver, the emulator will progress an indeterminate amount before being stopped by the framework.

IO-TRACE causes the I/O server to print the (pertinent) information from the call block and the returned status bits for each I/O call.

RESUME-STOP causes the PRIM framework to stop instead of resuming execution on emulator calls to MLP.STOP that would normally have resumed automatically, such as status stops and break stops whose breaktime programs cause resumption.

STOP-STOP causes the PRIM framework to stop on any emulator call to MLP.STOP, ignoring the parameter contained in R.37.

XOFF causes the PRIM framework to insert XOFF font-shifting and superscripting characters in the transcript file to distinguish control codes and user input from PRIM output. ASCII control codes are superscripted; PRIM outputs are switched to font A; and user inputs are switched to font B. The transcripts in each PRIM *User Guide*, the *User Reference Manual*, and this document were produced using the XOFF command with an A-font of FIX8 and a B-font of BOD9I.

Load loads a GPM binary file into the emulation context without first clearing it. MLP-900 registers (including auxiliary memory) can be loaded from GPM code (usually via a **constant** statement) that is assembled at an ORIGIN corresponding to that register's location in the swapped-out context. A map of the context appears in Section B.1. It is recommended that only registers containing constant values (*i.e.*, most of the mask registers) be specified in the GPM source.

No disables the various state-control flags that are set by the ENABLE command.

Save has the following additional subcommands for privileged users:

BREAKS
EMULATOR
REGISTERS-AND-AUX
TARGET-FORK

Tables loads an arbitrary descriptor-table relocatable file.

```
>tables (from file) ? File name
>TABLES (from file) <PRIM>U1050.DESCRIPTOR-TABLE;134cr
```

A.2 Debugger Commands

The additional debugger commands all involve the MLP-900; they fall in the execution-control, display, and storage categories. These commands are available only to users for whom whiz has been enabled. In each case, a single coded character effects the command:

Command	Coded Character
MLP break	↑B (*control-B*)
MLP step	-
MLP type	*
MLP change	/

MLP Break. Sets an execute breakpoint in the MLP-900. Only a single MLP breakpoint may be set at any time. To clear an MLP breakpoint, the command is entered without an argument.

```
#↑Break-MLP-at 5cr
#
```

MLP Step. Single-steps the MLP-900. Note in the examples below that, on each line in which an MLP-step is shown, the first hyphen was entered by the user and the balance of the line was completed by the debugger.

```
#--> MLP-step  to 16
#--> MLP-step  to 17
#--> MLP-step  to 21
#Go (to) cr
--> MLP-900 CM Address Compare at 5 Used 0:00.0 (MLP time)
#
```

MLP Type. Displays MLP-900 control memory symbolically. The output is the compiled code produced by the GPM compiler. Consecutive control memory locations are displayed until an *abort* intervention character is entered.

```
#/ 0cr

  0    0 BEAD   3 221      16  /IF TRUE THEN CALL 16 END;
  1    0 GENT   0 106  16 200  A.1216+0 ;
  2    0 MAST   4 221 221  16  /F.7 +NOT TRUE OR NOT TRUE ;
  3    0 MOVE   0   0   0 377  CE.0 +0(377);
  4    0 BEAD   3 221     261  /IF TRUE THEN CALL 261 END;
  5    0 GENT   0   0  37 200  R.37+0 ;
  6    0 BEAD   0   2      10  /IF NOT F.1 THEN GOTO 10 END;↑X
#
```

MLP Change. Permits GPM statements to be compiled into MLP control memory. The GPM statements entered for compilation must be terminated by an *escape*. The debugger prefixes these GPM statements with a dummy program title and ORIGIN statement and follows them with a semicolon and a program closing; the resulting "program" is then passed to the GPM compiler. The compiler's summary is displayed and its binary output file is loaded into the control memory image, replacing what was in the same locations. Note that the binary file is loaded even if the compiler detects errors.

```
#/ 6700esc
6700    0 GEAR   0   0   0   0  R.0 ←NOT R.0 OR R.0 (M.0);
6701    0 GEAR   0   0   0   0  R.0 ←NOT R.0 OR R.0 (M.0);↑X
#* 5esc   GPM: CALL 6700esc ...


%X      GPM.76.11.2     18-NOV-77 10:36:19 Pg 2 %
        %**No Errors Detected**%


#* 6700cr
GPM: CE.1←0;R.37←0;RETURNesc ...


%X      GPM.76.11.2     18-NOV-77 10:36:58 Pg 2 %
        %**No Errors Detected**%


#/ esc
6700    0 MOVE   0   0  20 377  CE.1 ←0(377);
6701    0 GENT   0   0  37 200  R.37←0 ;
6702    0 BORE  14 221 221   0  /IF NOT TRUE XOR NOT TRUE THEN GOTO +1 ELSE
 RETURN END;
6703    0 GEAR   0   0   0   0  R.0 ←NOT R.0 OR R.0 (M.0)↑X

#/ 5esc
   5    0 BEAD   3 221    6700  /IF TRUE THEN CALL 6700 END;
   6    0 BEAD   0   2      10  /IF NOT F.1 THEN GOTO 10 END;
   7    0 GEAR   6   0  37 204  R.37 ←R.37 OR 4 (M.0);↑X
#
```

For a whiz, the various space-access attributes in the emulator's descriptor tables (see Section 2.7.2) are ignored; all symbols (including the built-in symbols describing the MLP-900) are available and all meta-bits can be set in every space.

Appendix B
TENEX MLP Driver Interface

B.1 Control of an MLP-900 Process

A TENEX process (fork) can create and control an MLP-900 process (emulator) through the interface to the MLP driver in the TENEX monitor. The driver interface is implemented using a new device type known as "MLP:" and existing file and device JSYS's. The emulator's context is swapped into and out of a ten-page region in the fork itself; the emulator's main memory is mapped into a target fork that can be either the controlling fork or an inferior fork created for the purpose.

Since the target fork is directly accessed as the emulator's main memory, the fork can communicate with the running emulator through shared memory as well as through calls (MLP.CALL) and MLP action requests (F.130 through F.137). The context, however, is copied into the MLP-900 at each start/resume and back out at each stop; thus the fork can validly manipulate the context only when the emulator is stopped. The context region of the fork begins on a page boundary and has the following organization:

0 - 6777	control memory locations 0 - 6777
7000 - 7037	R.0 - R.37
7040 - 7057	M.0 - M.17
7060 - 7077	MISC.0 - MISC.17, excepting: 7074: MISC.36 (VADRC) 7075: MISC.37 (CMADRC)
7100 - 7157	(CE.0) - (CE.136), 16 bits per word, right justified: 7100 - 7117: (CE.0) *ff*. 7120 - 7123: (P.0) = (CE.40) *ff*. 7140 - 7157: S.0 = (CE.100) *ff*.
7160 - 7755	not used
7756 - 7777	control-memory locations 7756 - 7777
10000 - 11777	A.0 - A.1777

When the emulator stops, only the OE and CE registers and auxiliary memory are swapped out, since control memory cannot be altered by the emulator.

B.2 TENEX JSYS's Involving the MLP-900

A JFN obtained for the MLP-900 with the GTJFN JSYS is then used in the following JSYS's:

- OPENF opens the JFN; it must be performed before any other JSYS using that JFN. Use byte size of 36, read access, and mode of zero.
- CLOSF closes and releases the JFN.

- MTOPR controls the emulator. Four operations are defined:
  - 1 define interrupt channels
    - AC3: 0-5 emulator STOP PSI channel (>36 for no interrupt)
    - 6-11 emulator CALL PSI channel (>36 for no interrupt)
    - 12-36 not used
  - 2 halt emulator and swap out
  - 3 start/resume emulator
    - AC3: 0-17 context address
    - 18-35 target fork handle
  - 4 interrupt emulator (send action request)
    - AC3: 10-17 mask
    - 28-35 bits
- BIN reads next call parameter-word from the call buffer (waits if none available)
- SIBE skips if the call buffer is empty
- BKJFN does not work for "MLP:"
- GDSTS returns status of the emulation process in AC2, AC3, and AC4:
  - AC2: 0-17 status; 0 = running
    18-35 emulator micro-PC
  - AC3: action requests pending in the driver (not from F.130-F.137)
  - AC4: total MLP time (milliseconds)

The AC2 bits describe the reason(s) the emulator is stopped. If the emulator is currently in the driver's run queue, it appears to be running (status 0). The bits are

B0-B5	reserved
B6	supervisor facility violation (action request)
B7	protection violation (action request)
B8	virtual address compare (action request)
B9	control memory compare (action request)
B10	extended stack overflow
B11	not used
B12	MLP.STOP call (also set whenever any of B6-B11 are set)
B13	MLP hard error (probably fatal)
B14	MLP soft error (probably did no damage, but no guarantees)
B15	frozen (the call buffer is full)
B16	illegal memory reference (no targer fork or protected page)
B17	halted (by "MTOPR 2")

The driver keeps a word containing pending action requests that have not yet been inserted into the MLP flops F.130 through F.137. This word is returned by the GDSTS JSYS in AC3 and is altered by AC3 of an "MTOPR 4" JSYS as follows: when the mask is zero, the bits are OR'ed into the driver's pending word; when the mask is not zero, the pending bits that are masked-in are loaded from the corresponding bits in AC3. Note that only action requests that are still pending in the driver can be cleared; once transferred to the MLP flops they cannot be reset by the fork except by halting the emulator and clearing the bits in the context.

Appendix C
GPM Reserved Words

An alphabetic list of GPM reserved words follows. Equivalent forms are shown in parentheses.

A.0-1777 (OE.2000)
A.PG.0-3 (OE.PG.4)
AERR (F.111)
AND
ARL.1-4 (F.170)
ARL.5 (F.150)

B.0-3
BEGIN
BERR (F.112)
BLOT.0-7
BREAK
BY

CALL
CASE
CCP (F.307)
CE.0-377
CED.0-177
CKC (F.164)
CKT (F.166)
CMADR (F.110)
COF.1-2 (F.140)
COMMENT
COP (F.300)

DATAI (OE.1033)
DATAO (OE.1032)
DECREMENT
DIVIDE
DO (DO.BEGIN)

ELSE
END
ENTRY
EPAR (F.103)
EQUATE
ERS (F.340)

F.0-377
FALSE
FINISH
FOP
FSI.0-1 (F.376)

GOTO

H.0-1
HEXADECIMAL.CODE

IF
INCREMENT
INDIRECT.0-1
INTO (INTO.BEGIN)
ITRAC (F.153)

LABEL.TABLE
LEFT

M.0-17
MBS (F.167)
MINUS
MISC.0-37 (OE.1000)
MMERR (F.116)
MOD.0-1 (F.174)
MODE
MOE
MULTIPLY

NAMED
NORMAL.CODE
NOT

OE.0-7777
OE.PG.0-17
OP.0-17
OPAR (F.102)
OR
ORIGIN
OSI.0-3 (F.354)

P.0-17
PAGE (F.121)
PANIC (F.101)
PERR (F.113)
PIR (OE.1004)
PLUS
POWER (F.100)
PRINTOFF
PRINTON
PROT (F.123)

R.0-37
RCM
RETURN
RIGHT
ROW
RSB

S.0-17 (CED.40)
SAD
SARM.0-1 (F.160)
SHD (F.353)
SHE (F.145)
SHIFT.0-10
SHIFT.DUAL.L
SHIFT.EO.L
SHIFT.ER.L
SHIFT.OE.C
SHIFT.OE.L
SHIFT.RE.C
SHIFT.RE.L
SHIFT.SINGLE.L
SIR (OE.1005)
SLBC.0-17 (F.60)
SOF (F.147)
SOP
SOS (F.146)
SOVF (F.104)
SSW.0-7 (F.340)
SUNF (F.105)
SUPVCT (F.177)
SUPVF (F.122)
SUPVLB (F.176)
SWITCHON

TASK (F.120)
TEMPORARY
THEN (THEN.BEGIN)
THRU
THZ (F.304)
TITLE
TRAC (F.130)
TRBY (F.165)
TRUE
TSI.0-1 (F.374)

UOVF (F.106)
UUNF (F.107)

VADR (F.124)

WAR (F.305)
WBP
WCM
WOP
WOS
WSB

XBUS (OE.4000)
XBUS.0-3
XLATOR.0-777 (OE.4400)
XLATOR.PG.0-1 (OE.PG.11)
XOR

ZRF.1-2 (F.142)
ZSI.0-7 (F.360)
ZSP (F.301)

References

1. Bobrow, D. G., J. D. Burch, D. L. Murphy, and R. L. Tomlinson, "TENEX, A Paged Time-Sharing System for the PDP-10," *Communications of the ACM*, Vol. 15, No. 3, March 1972, pp. 135-143.

2. Meyer, T. H., J. R. Barnaby, and W. W. Plummer, *TENEX Executive Language Manual for Users*, Bolt Beranek and Newman, Inc., Cambridge, Mass., April 1973.

3. *MLP-900 Multilingual Processor--Principles of Operation*, STANDARD Computer Corporation, Santa Ana, Calif., 1970.

3. *DECsystem-10 Assembly Language Handbook*, Digital Equipment Corporation, Maynard, Mass., 1972.

5. *TENEX User's Guide*, Bolt, Beranek and Newman, Inc., Cambridge, Mass., January 1973.

Index

PRIM SYSTEM: USER REFERENCE MANUAL

Contents

PRIM SYSTEM: USER REFERENCE MANUAL

INTRODUCTION

This document is the common reference manual for all users of the PRIM system, both those using one of the existing emulation tools and those writing new emulators. For the former, this manual is supplemented by the appropriate tool-specific guide (*e.g., PRIM System: U1050 User Guide*); for the emulator writer, the supplement is *PRIM System: Tool Builder Manual*.

The PRIM system is always in one of three states, known as the exec, the debugger, and the target execution states. The transition between states is controlled by the user. Both of the first two states are PRIM command processors that take commands from the user and execute them. The exec, whose command prompt character is ">", is used principally for setting up a target environment; the debugger, whose command prompt is "#", is used for the detailed examination and control of the executing target machine. Target execution includes the emulation of not only the CPU, but also clocks and assorted peripheral IO devices. The three sections following the introduction describe each of the states in turn.

The PRIM exec and debugger commands are illustrated with examples taken from actual session transcripts. In all the examples, user input is *italicized* to distinguish it from PRIM output. Input control characters appear as their abbreviations superscripted (*e.g.,* ^{esc}).

GENERAL INPUT CONVENTIONS

User input to PRIM, both exec and debugger, is generally free-format and case-independent. Leading spaces and tabs are ignored, and lower case is treated as its upper case equivalent (except in quoted strings, where case is potentially significant). User input to the target machine during target execution state is in the format required by the target system.

Certain characters have been assigned editing and intervention functions when input by the user. The editing characters apply only to the PRIM exec and debugger, while the intervention characters apply to the target execution state as well. The specific characters assigned to most of the functions may be altered (via the exec Change command) to suit one's needs. The editing functions are valid at any time during PRIM command input; commands are not executed until after the final character has been accepted.

Back-space (cntl-H) erases a character from the current word or term of input. The back-space is echoed as a backslash (\) followed by the erased character. When there are no erasable characters, a bell (cntl-G) is echoed instead.

Alternate back-space (initially cntl-A) performs a function identical to ***back-space***; it is provided as a convenience.

Backup (initially cntl-W) erases the current word or term of input. It is echoed as backslash (\) followed by the first character of the erased word.

Retype (initially cntl-R) retypes the current input line; it is useful after a confusing amount of editing has occurred.

Delete (initially DEL or RUBOUT) aborts the current input command or subcommand, allowing the user to re-enter it. It is echoed as " XXX".

Question (?), when entered at the beginning of a command field, elicits a description of the expected input, followed by a retype of the line. When the expected input is a selection from a list (or menu), the entire list is shown.

The intervention characters are valid at any time, including command input, command interpretation, and target execution.

Abort (initially cntl-X) interrupts the current activity and returns control to the command level of either exec or debugger. When used to cancel an exec or debugger command, control returns to the top level of the same state; abort is the only means of canceling a command when the user is in subcommand mode. When used to interrupt target execution, control returns to the state from which execution was initiated; abort is the only means of stopping a looping target machine.

Status (initially cntl-S) produces a one-line summary of target machine status, including program counter, emulated elapsed time, and active IO devices. The command is valid at any time, but useful primarily in execution state.

The following character is active only during target execution.

Control-shift (initially cntl-1) permits the user to enter (during execution) a control code that cannot be entered directly because it is intercepted by either PRIM or the operating system; the PRIM characters inolved are ***status***, ***abort***, and ***control-shift*** itself. The next ASCII character following the ***control-shift*** (other than the digits 0 thru 9) has its two leading bits cleared, thus converting it to an ASCII control code (*A* or *a* to *cntl-A*, etc.). *Control-shift* followed by a digit results in an input that is outside the normal target character set and is used for particular target-machine-dependent functions. The control-shift character itself is not echoed, and not passed to the target machine. If execution terminates before that next character is input to the target device, the control-shift is canceled; it is not retained for the next resumption of execution.

PRIM EXEC

The PRIM exec is the initial state of a PRIM session. Exec commands are concerned primarily with building target configurations, saving PRIM session results, restoring previously saved sessions, and accessing or creating files (within the file space of the host operating system).

The exec prompt character is ">", indicating that PRIM is in exec state and that the exec is awaiting a new command; it is always shown on a new line. Individual input fields consist of keywords (a word selected from a menu), decimal numbers, and file names. Exec commands are composed of fixed sequences of fields, each terminated by a delimiter character; a final confirmation consisting of a ***return*** is often required.

Keywords are selected by any unambiguous leading substring. Often, a single character suffices; three characters are always sufficient. Numbers are specified in their entirety. File names are specified according to the conventions of the operating system. All commands that will use a file for output require the name of a new file (except the Mount-Append and Mount-Old commands, which modify existing files); all other file commands require the name of an existing file. In TENEX, an existing file name -- and a new file that is a new version of an existing file name -- is recognized (and completed) in response to an input ***escape***.

The normal delimiters that terminate command fields are ***return***, ***escape***, and ***space***. ***Escape*** and ***space*** function identically except that the former generates feedback to the user while the latter generates none; the feedback produced by ***escape*** includes both field completion and next-field prompting (which is given in parentheses). ***Return*** is used to complete a command immediately, bypassing any remaining fields and confirmation; if further input is required, the ***return*** is treated as an ***escape***. (In the examples that follow, ***escape*** termination is used to show the prompts.)

Keywords that involve either devices or parameters are machine-dependent; the selections shown in the examples are meant to be illustrative rather than definitive. Device specification is further complicated when two (or more) of the same generic device are installed. Therefore, for device names, two further delimiters are utilized, ***at*** ("@") and ***colon*** (":"). A fully qualified device name consists of ***generic-name @ channel-number : unit-number***; the numbers are required only to the extent necessary to specify a particular device. When a device name is terminated by one of the standard terminators, and when further disambiguation is required, the exec prompts explicitly regardless of the terminator.

The remainder of this section consists of the descriptions of the exec commands in alphabetical order. Each command description begins with a transcript showing one or more examples of the command and its various options. Those commands that require a second keyword show that list via an input ***question***. The exec commands are:

```
>? One of the following:
 CANCEL
 CHANGE
 CLOSE
 COMMANDS
 DEBUG
 FILESTATUS
 GO
 INSTALL
 MOUNT
 NEWS
 PERIPHERALS
 QUIT
 REASSIGN
 RESTORE
 REWIND
 SAVE
 SET
 SHOW
 SYMBOLS
 TIME
 TRANSCRIPT
 UNINSTALL
 UNMOUNT
>
```

Comment.

```
>; this line is a comment<cr>
>
```

Any line beginning with a *semicolon* is treated as a comment. Comments are recorded in the transcript if one is open (see Transcript command).

Cancel abandons all outstanding IO operations for a designated device.

```
>ca<esc>NCEL (IO for device) ta<esc>PE-UNIT <cr>
>
```

This command is intended for use when, after an IO error halt (described in the section on target execution), the user wishes to abandon the device operation rather than mount a file and retry the operation. The list of outstanding IO operations, by device, is part of the Peripherals command output.

Change reassigns the PRIM control functions.

```
>ch^[ANGE (input code for) ? One of the following:
 ABORT
 ALT-BACKSPACE
 BACKUP
 DELETE
 RETYPE
 STATUS
 CONTROL-SHIFT
>CHANGE (input code for) ab^[ORT (from ↑X to) ? A Control Code.
>CHANGE (input code for) ABORT (from ↑X to) ↑P cr

>ch^[ANGE (input code for) d^[ELETE (from <DEL> to) ^[ [not changed]
>
```

This command allows the user to change the ASCII control code assigned to any of the listed PRIM control functions from its current assignment to another (currently unassigned) control character. The function name is the second word of the command; when it is terminated with an *escape*, the current assignment is noted in the noise. The entire set of ASCII control codes (including *delete*) is available excepting *null*, *back-space*, *line-feed*, *return*, *escape*, and *unit-separator* (TENEX *end-of-line*) which have fixed functions in PRIM. For *abort* and *status* the set is limited to *cntl-A* thru *cntl-Z*.

Close terminates the current transcript file if one is open.

```
>cl^[OSE (transcript file.) cr
>
```

A transcript file is opened using the Transcript command; it is automatically closed at the end of a session.

Commands redirects subsequent input from a file.

```
>co^[MMANDS (from file) command.file^[ cr
>
```

This command causes PRIM to read its subsequent command input from the named file instead of the user terminal (or current command file). The file input is treated exactly as terminal input except that intervention functions (*abort* and *status*) are valid only from the terminal. Should a command in the file cause execution to be resumed, input that normally would come from the user terminal is taken instead from the file. Input reverts to the previous source at the end of the file; an *abort* terminates all command files and reverts input to the user terminal. Command files may be nested. Command files are very useful for common session-initialization sequences.

Debug transfers control to the PRIM debugger.

```
>d^[EBUG
#return (to EXEC) cr
>
```

The PRIM debugger is described in the next section; control is returned to the exec via the debug Return command.

Filestatus returns information about mounted files for all or designated devices.

```
>fESCILESTATUS (for device) ESC ALL
 Record   File Name          Device
 12       CARD.DECK          CARD-READER
 12       User Tty           PRINTER
 825      TERMINAL.INPUT     TERMINAL (In)
 12345    TERM.OUT           TERMINAL (Out)
 2+6      ABCD.EFG           TAPE-UNIT:0

>fESCILESTATUS (for device) caESCRD-READER
 Record   Type  Byte/Last     File Name
 12       Bin12 960/1280     CARD.DECK
>
```

When the device field is empty (*return* or *escape*) all mounted files are listed; otherwise just the file(s) on the named device are listed. The latter case gives more complete status than does the former. The output fields are:

Record tells the current position of the device or the number of records which have been processed. For disks, it is a sector number; for card readers and punches, a card count; for communication lines, the total number of bytes transferred; for mag tape units, the position from beginning of tape expressed as files + records.

File Name is the name of the file; the name "User Tty" is displayed when *THIS-TERMINAL* is the file.

Device is the emulated device on which the file is mounted.

Type describes the type of file, either Ascii or Binxx, where xx is the file byte size. The type may have been explicitly specified at mount time, or it may have been assumed by PRIM.

Byte/Last is, for a mounted disk file, the current byte position in the file and the total number of bytes in the file.

The marginal notation "[not opened]" indicates that the named file could not be found (this occurs only to a restored file) and that the device must be reassigned to another file (or to the same file via a new path name).

Go transfers control to the target execution state.

```
>gESC0 (from 1234) cr
--> MACHINE running at 5670, Used 0:00.4
--> MACHINE halted at 6543, Used 0:01.0
>
```

This command transfers control from the PRIM exec to the emulator or target machine, in its current state. Control returns to the exec when the target machine halts or a breakpoint is encountered (see the debugger Break command) or the user interrupts execution with an *abort*.

In the example, the user followed the command with a *status* request (the *status* character itself is not echoed) resulting in the first reply line (MACHINE running at ...); the target machine is still running. Eventually the target machine halted, producing the second status line and returning control to the exec as evidenced by the exec prompt.

Install adds a designated type of device to the machine configuration.

```
>iesc NSTALL (device) ? One of the following:
 CARD-READER
 PRINTER
 TAPE-CONTROLLER
 TERMINAL
>INSTALL (device) pesc RINTER (CHANNEL) 1esc
>>? SPEED
>>aesc PEED (characters per second) esc300
>>cr

>iesc NSTALL (device) taesc PE-CONTROLLER (CHANNEL) 3esc cr
How many TAPE-UNIT's do you want? 2cr
For the first TAPE-UNIT, (UNIT) 0esc cr
>>cr
For the second TAPE-UNIT, (UNIT) 1cr
>>cr
>
```

The device type is selected from among those implemented. The user is prompted for each necessary item of information, typically including an address for the device in the target IO address space and the number of units to install. After the required information is gathered, sub-command mode (">>" prompt) is entered to gather optional parameters; any optional parameter not supplied takes on its default value. Subcommands are terminated by an empty command, *return* only. An installed device is initially unmounted -- there is no file associated with the device for purposes of actual IO.

When the device being installed is a multi-unit controller, the dialogue proceeds through each of the individual units to gather their parameters. After the command is completed, the controller is no longer visible; only the individual units are. An *abort* aborts the entire command, not just the current unit.

Installation is permitted only before any execution has taken place. Typically, a user or user group installs a standard configuration and then saves it for use in all subsequent sessions (see the Save-Configuration and Restore commands). The optional parameters of an installed device may be changed at any time using the Set command.

Mount associates a file with an installed device.

```
>m[esc]OUNT (A,I,N,OL,OU,T,?) ? One of the following:
 APPEND
 INPUT
 NEW
 OLD
 OUTPUT
 THIS-TERMINAL
>MOUNT (A,I,N,OL,OU,T,?) t[esc]HIS-TERMINAL  (on device) p[esc]RINTER cr

>m[esc]OUNT (A,I,N,OL,OU,T,?) n[esc]EW (in & out file) /ABCD.EFG;1[esc] (on device)
 ta[esc]PE-UNIT cr
>

>m i[esc]NPUT (from file) card.deck[esc] (on device) ca[esc]RD-READER cr
>>? BINARY or ASCII
>>b[esc]INARY (with byte size) 12cr
>>cr
>
```

Associating a file with an installed device causes subsequent emulated IO for that device to be directed to the file. The second keyword following Mount determines the direction of data flow and the choice of an old (existing) or new file. A file must be mounted on a device before any actual IO can take place.

APPEND mounts an old file for output only, with the subsequent output being appended to the previous contents of the file.

INPUT mounts an old file for input only.

NEW mounts a new file for both input and output (the file is initially empty).

OLD mounts an old file for both input and output (subsequent output overwrites any existing file data).

OUT mounts a new file for output only. For a disk or tape device, OUT is treated as NEW.

THIS-TERMINAL associates the user terminal -- instead of a named file -- with the named device. The mounting is for both input and output unless a file has already been mounted for one, in which case the terminal is mounted only for the other. The terminal is known to be an ASCII "file". The terminal may be mounted only once for input; it may be mounted for output (or on an output-only device) any number of times, but the output is not labeled as to source.

Only some of the forms above are applicable to any given device. For a disk- or tape-like device, an INPUT, OLD, or NEW file is expected; an OLD file is one that was NEW in a previous PRIM session, and is being re-used, while an INPUT file is an old read-only file. For a bidirectional communication device (*e.g.*, a terminal), two files are required: an INPUT file and either an OUTPUT or APPEND file. Alternatively, a real terminal may be used for both (or either one). For an input-only device, INPUT and OLD are identical; for an output-only device, OUT and NEW are identical.

For those devices that deal exclusively with character data, the mounted file is always taken as an ASCII text file; character translation is performed as part of the IO process. (This allows the file to be created and/or processed by any operating system utility that deals with text files.) For tape and disk devices, the file format is internal to PRIM (and therefore not requested from the user); the data is recorded directly. For other devices the user is asked, via subcommand mode (">>" prompt), whether the mounted file (NOT the device) is an ASCII text file or a binary file containing a stream of pure data in bytes of some fixed size. The default is a binary file of a device-dependent byte size.

Once a file has been mounted on a device, all exec commands that refer to the file require the device name as the specifier; for communication devices, where two files are normally mounted, the device name is followed by a direction selector. The file name itself is not used as the internal identifier.

News reads the PRIM on-line news file.

```
>n^escEWS
 Do you want to see  4-APR-77  Changes in PRIM ?: ^escYES
  [  Here comes the message regarding changes of 4-APR-77 ...  ]
 Do you want to see 24-MAR-77  Preliminary Documentation ?: del XXX
>
```

The date of the most recent news message is shown automatically at the start of each session. In response to the command, each message's date and subject is shown, beginning with the most recent message. For each message, the body may be seen (*YES*) or skipped (*N()*), or the command may be terminated (*delete* or *abort*).

Peripherals returns information about the installed devices.

```
>p^escERIPHERALS
Chan  Unit  Mounted  Device
 1     0      No     PRINTER
 2     0      Yes    TERMINAL
 3     0      Yes    TAPE-UNIT
 3     1      Yes    TAPE-UNIT

 active devices: TERMINAL
>
```

This command produces a listing of all the installed devices, together with their IO addresses and a notation concerning whether they have files mounted. It also lists all devices which have suspended IO operations. Ordinarily, suspended operations are limited to (1) IO error conditions and (2) input operations where the input file is a real terminal and no input was available when target execution stopped.

Quit terminates a PRIM session.

```
>q^escUIT
  Quitting MACHINE  [Confirm] cr
e
```

Terminating the PRIM session involves closing all open files and returning control to the process that initiated the PRIM session. The session cannot be continued.

Reassign specifies a new file for a mounted device.

```
>reaASSIGN (device) taPE-UNIT (to file) new.fileesc cr
>
```

This command is used to substitute a new file specification when, after a prior Restore command, a previously mounted file cannot be found. In particular, a restore done from a different directory than the one in force at save time has trouble finding any of the mounted files. Reassign may only be used for devices/files that are marked "[not opened]" in a file-status display. The new file is assumed to have the same characteristics as the old one and is positioned at the same file position.

Restore recovers the state information saved in a file.

```
>resTORE (from SAVE file) /ABCD.CONFIG;1esc cr
  restored CONFIGURATION from TUESDAY, MAY 3, 1977 12:35:08 PDT
>
```

The current context is updated with the complete or partial environment previously saved in the designated file by the Save command. For the addressable regions -- machine memory, registers, etc. - the saved data replaces the current data only for those cells that were actually saved; cells not saved are not cleared. (Thus, nonoverlapping memory images are merged.) For nonaddressable regions -- symbol, configuration, and breakpoint -- each one is completely replaced if present in the file. The date and region(s) saved are shown, followed by a list of any mounted files that cannot be found.

Rewind returns a device's mounted file(s) to the beginning.

```
>rewIND (device) taPE-UNIT cr

>rew terMINAL (B,I,O,?) ? One of the following:
 BOTH
 INPUT
 OUTPUT
>REW TERMINAL IesNPUT cr
>
```

This command is useful for retrying a program without unmounting and remounting files. (Files are always rewound when mounted, except for Append files, which cannot be rewound.) For a terminal-like device that requires separate input and output files, the user optionally specifies which file is to be rewound; the default is *BOTH*.

Save copies selected state information into a file.

```
>saVE ? One of the following:
 ALL
 CONFIGURATION
 FORMATS
 MEMORY
 SYMBOLS
>SAVE eNFIGURATION (on file) aBCD.CONFIG;1cr
>
```

This command saves on the (new) file an image of the region(s) selected for saving. The contents of the file can later be restored for use in this or another session. The second word of the command selects one of the save options.

ALL saves everything -- a complete checkpoint of the target machine and debugging state. "Everything" includes memory, all addressable registers, installed devices, mounted files together with their positions, debug breakpoints and their programs, debug formats and modes, defined symbols, and the internal state of the emulated machine.

CONFIGURATION saves all the machine configuration data, including installed devices, mounted files (if any), machine parameters, and debug formats and modes. This command is allowed only before any execution takes place. Useful for creating a standard machine configuration (possibly with some standard files mounted) for use in subsequent sessions.

FORMATS saves all the formats that have been defined (using the debugger Format command).

MEMORY saves those regions of the machine memory that are not clear. (At the start of a PRIM session, memory is already cleared.)

SYMBOLS saves all the user-defined symbols, both those loaded via the exec Symbols command and those defined directly via the debugger New-symbols command. The file that results is a SAVE/RESTORE file, not a SYMBOLS file!

Set changes the values of user-settable parameters.

```
>se^escT (<empty> or device) cr
>>? One of the following:
 CLOCK
 MEMORY
 SPEED
>>c^escLOCK (ticks per second) ^esc1000 cr
>>m^escEMORY (8K modules) 4cr
>>cr

>se^escT (<empty> or device) p^escRINTER
>>s^escPEED (characters per second) 150cr
>>cr
>
```

Following the command word, the user selects the group of parameters he wishes to alter. An immediate *return* selects the global machine parameters; a device name selects the parameters of that particular installed device (the parameters of multiple installed instances of the same device type need not have identical settings).

Any number of parameters from the selected group may be changed. In response to the subcommand prompt (">>"), the name of a parameter and its new value are entered; each change is made immediately and a new subcommand prompt appears. The command is terminated by an empty input, *return* only, or by an *abort* (which does not undo any parameters previously changed). The list of possible parameters is highly machine- and device-dependent; it typically includes the size of memory and the speed of each device.

The value of a parameter is either a (decimal) number or a keyword from a parameter-specific list; a *question* in the value field reveals which is expected. An *escape* sets the parameter to its default value.

Show displays the values of all the parameters in a group.

```
>sh^escOW (<empty> or device) cr
 CLOCK is 1000 ticks per second
 MEMORY is 4 8K modules
 SPEED is 750 nanoseconds per memory cycle

>sh^escOW (<empty> or device) p^escRINTER
 SPEED is 200 characters per second
>
```

Following the command word, the user selects either the global machine parameters (*return*) or the parameters of an installed device. The names and current values of all the parameters are displayed.

Symbols reads an ASCII symbol-table file.

```
>sy^escMBOLS (from file) SYMBOLS.EXAMPLE^esc cr
>
```

This command causes PRIM to build a user-defined symbol table from the data in the named file, which is a structured ASCII text file. The file may define values for both global symbols and program-local symbols that are organized into programs. In the PRIM debugger, the global symbols plus the local symbols of the currently open program are accessible at any time. Symbol values in the file are octal. The form "name == value" defines a global symbol; the form "name = value" defines a local symbol; the form "name:" establishes a program name to which subsequent local symbols are assigned. The file is free-format in that spaces, tabs, commas, and new-lines may occur anywhere -- except in the middle of names or values. The following is a sample symbols file.

```
ALPHA--45
BETA==12345
PR1:  A=2000, B=2132, C = 2241
XYZ:
A-3212   AA=3245, AAA=3261,AAAA=7777
```

Symbol files are intended to support the moving of symbolic label data from an assembler or linking loader into PRIM for use in symbolic debugging.

Time displays time-of-day and time-used information.

```
>ti^escME (is) TUESDAY, MAY 3, 1977 12:34:33-PDT
 Used 0:14.6 PRIM time;  Used 0:02.7 MLP time.
>
```

This command displays the date, time of day, the amount of PRIM time used and the amount of MLP-900 time used in this PRIM session. (Elapsed target machine time is displayed in response to *status*.)

Transcript transcribes the subsequent PRIM session on a new file.

```
>tr^escANSCRIPT (to file) new.file^esc cr
>
```

All transactions with the user terminal, including execution-time IO to THIS-TERMINAL, is transcribed until either the user terminates the session (with a Quit command) or closes the transcript. Only one transcript may be open at a time. A header line containing the date and time is placed at the head of the file.

Uninstall removes an installed device.

```
>uni<esc>NSTALL (device) ? PRINTER or TAPE-UNIT
>UNINSTALL (device) ta<esc>PE-UNIT (unit): 1<esc> cr
>
```

This command is the inverse of the Install command; it removes an installed device from the configuration, first unmounting its files if necessary.

Unmount unmounts the file(s) from a device.

```
>unm<esc>OUNT (device) p<esc>RINTER cr

>unm ter<esc>MINAL (B,I,O,?) ? One of the following:
 BOTH
 INPUT
 OUTPUT
>UNM TERMINAL <esc> BOTH cr
>
```

The unmounted file(s) are closed. For a terminal-like device that requires separate input and output files, the user optionally specifies which file is to be unmounted; the default is BOTH.

PRIM DEBUGGER

The PRIM debugger is a table-driven, target-machine-independent, interactive program for debugging a PRIM emulator or a target program running on such an emulator. It is tailored to a specific target machine by tables prepared as part of an emulation tool. Basically, it permits a user to set and clear breakpoints and to examine, modify, and monitor target system locations. Target system assembly language and symbolic names are recognized, and arithmetic is performed according to the conventions of the target machine. The debugger command prompt character is "#"; each level of subcommand adds another "#" to the prompt.

ARGUMENTS

Most debugger commands take arguments in the form of values, expressions, expression-ranges, lists of expressions, or lists of expression-ranges as defined below.

Values

A value is an assembly-language instruction, a form, text, or an expression-list. Assembly language instructions are parsed by a table-driven assembler/disassembler that accepts the same syntax as the assembler for the target machine. User symbols will be recognized if they have been supplied in user symbol-table files (see the exec Symbols command) or have been declared individually (see the debugger New-symbol command).

A form requires that the user previously define a corresponding format (see the debugger Format command). A form is represented by the format name followed by an expression-list, as in the following example.

```
F1 0, 7, 3
```

Text is represented as a double-quote ("), followed by an arbitrary delimiter character, followed by a sequence of other (non-delimiter) characters, followed by another occurrence of the delimiter character, as in the following example.

```
"/This is text./
```

Expressions

An expression is any well-formed sequence of constants and symbols that are defined for the target machine; the symbols (which are machine-specific) may represent either locations or operators whose rules of combination determine what is a well-formed expression. A location symbol may represent a named hardware element or a globally or locally defined user location. An operator may either be unary (preceding its operand) or binary (coming between its operands in infix notation). The precedence of operators is a function of the target machine, except that all unary operators are assumed to have the same precedence value, which is higher (more strongly binding) than that for any binary operator. If brackets are permitted (*e.g.*, parentheses), their precedence value is higher than that of unary operators. For example, A-B and -B+A will evaluate the same, but will differ from -(B+A), which will evaluate the same as -B-A. A bracketed subexpression may itself attain the full complexity of an expression. The behavior of operators is machine-specific.

Expression ranges

An expression-range consists of the triple: expression (lower bound), colon, expression (upper bound). It represents a sequence of locations starting at the lower bound and continuing through successive locations to include the upper bound. The upper bound may not be less than the lower bound. Wherever an expression-range is allowed, a single expression is accepted and treated as if it had been entered as both the lower and upper bounds of a range. If the two bounds in a range address different spaces (see the discussion of Spaces below) within the target machine, the sequence of locations is restricted to that space addressed by the lower bound. Two special forms of expression ranges are recognized. If the second expression in a range is "*-1*", it is treated as being the largest address in the space referenced by the first expression. If the second expression in a range is of the form "*+ expression*", it is treated as if it were "*(lower bound) + expression.*"

Lists of expressions or ranges

A list of expressions consists of at least one expression, followed, optionally, by any number of occurrences of a comma followed by an expression. A list of expression-ranges has the corresponding structure of at least one range, followed, optionally, by any number of occurrences of a comma followed by a range. An example of a list of ranges is

0:10, 20, 30:50

Note that the second element of the list (20) is an example of a range with a defaulted upper bound.

SPACES

Addressable locations in a target system are organized into constructs called spaces. A space consists of a set of addressable locations that is closed under a successor function and its inverse (a predecessor function). For example, main memory constitutes a space, typically starting at location zero and continuing through an arbitrary number of locations. The successor to the last element of a space is the first element in that space; and the predecessor of the first element is the last one. In some cases, machine locations are grouped into a space for convenience, even when the concept of a successor function for elements of that space has no correspondence in the actual target system. Such a space might consist of testable indicators. The machine symbols are identified in the tool-specific user guide.

For purposes of the debugger, every addressable location in a target system is represented by a pair: (*space, element*). When a range is specified, two such pairs $(a,b):(c,d)$ are implied. To avoid ambiguities where a and c differ, the debugger ignores c and treats such a range as a sequence of locations, all in space a, starting with element b and continuing through element d.

SYNTACTIC UNITS

The basic syntactic units the debugger deals with are

1. Literals
2. Symbols
3. Punctuation

Literals

Literals are character constants, numeric constants, or single characters that have some encoded meaning (which may be context-dependent). A character constant is supplied to the debugger as a machine-specific character-constant prefix string followed by a string of data characters of arbitrary length, followed by a machine-specific character-constant suffix string of the general form:

prefix-string character-data-string suffix-string.

If the first character of the suffix string is to be included in the data string, it must appear doubled. Character constants are converted to binary (right justified) and are truncated to fit the element in question. As the form of a character constant is machine-specific, it is described in the tool-specific user guide.

A numeric constant is supplied to the debugger as a machine-specific (and optional) radix-prefix string followed by a string of digit characters followed by a machine-specific (and optional) radix-suffix string of the general form:

prefix-string digit-string suffix-string

The prefix and suffix strings establish the radix within which the digit characters are evaluated. The digit characters for any radix *r* are the first *r* characters of the set {0,...,9,A,...,Z}.

Coded characters have independent meaning only within certain contexts: at appropriate points in the dialogue they designate a particular debugger command, a mode, a breakpoint type, etc.

Symbols

There are five types of symbols: machine symbols that are assigned to hardware elements in the target machine, predefined opcodes for symbolic instructions, user-supplied names of formals, operators for expressions, and user symbols that can be assigned to arbitrary memory locations. Machine symbols are given in the tool-specific user guide; other symbols are assumed to be familiar to the user.

User symbols are either loaded from a file using the exec Symbols command or individually defined using the debugger new-symbol command. The symbols include both global symbols and program-local symbols that belong to specific named programs. The global symbols are available at all times; the program-local ones only when theirs is the open local symbol table.

Punctuation

Punctuation marks are characters with a predefined syntactic (and usually semantic) role. The punctuation characters are the separators (*comma* and, in format definitions, *space*), the terminators (*return*, *escape*, and, in replacement operators, *back-slash* and *up-arrow*), and a semantics-free delimiter (*space*). *Escape* is used as a terminator instead of *return* to invoke a subcommand or an additional feature of a command (*e.g.*, in Mode or Breakpoint commands described below).

ERROR DETECTION AND EDITING

Debugger commands are examined for errors as they are entered, character by character. As soon as an error has been detected, a bell (beep) is echoed and further input is rejected, except for the generic editing characters ***back-space***, ***retype***, ***backup***, ***delete***, or ***abort***.

COMMANDS

Debugger commands are all single characters; they can be organized into several groups: debugger control, execution control, display, and storage. Each is listed below. Unless otherwise indicated, the command character is the first character of the command name.

Debugger Control

Debugger Control commands provide for user control over several aspects of the behavior of the debugger. They permit the user to execute commands indirectly or conditionally, to return from the debugger to the PRIM exec, and to control the debugger's representation of data. The Debugger Control commands are:

Use. Calls a designated break-time program as if some breakpoint associated with that program had just occurred. A program number must be designated that corresponds to an existing break-time program. Program numbers are shown when the breakpoint data base is displayed (see the break command); the program itself can be seen using the program-edit command.

```
#Use-program ?(number of an existing break program)
#Use-program 2cr
```

If the use command is itself in a break-time program, then a go command executed in the called program causes termination of the calling program as well as of the called program.

If. Tests the supplied expression and, if it is true, executes the following subcommand. A true expression is one whose value is ***odd***; relational operators yield a value of one when true and zero when false. The tested expression must be terminated by an ***escape***.

```
#If ?(expression)
If 3esc <then> ##Type 0cr
 00: 00   #

#If 2esc <then> ##Type 0cr
#
```

Return. Returns control to the PRIM exec; confirmation is required.

```
#Return (to EXEC) cr
>
```

Mode. Interrogates default and current modes and changes modes. A *question* after the command character *M* will elicit the default and current mode setting; another *question* will list all mode settings and associated mode-code-characters.

```
#Mode ?
Current and (Default) mode settings:
  Feedback        Verbose        (Verbose)
  Output          Bits           (Bits)
  Addresses       Symbolic       (Symbolic)
  Line-format     Dense          (Dense)
  Radix           8              (8)
Type ? for more

#Mode ?
Feedback:
  C     Concise
  V     Verbose
Output:
  B     Bits
  F     Formatted (format-name)
  I     Instruction
  N     Numeric
  T     Text

Addresses:
  A     Absolute
  S     Symbolic
Line-format:
  D     Dense
  E     Expanded
Radix:
  Rn    Radix-base n (1 < n < 37 decimal)
#
```

A list of mode settings is expected following the Mode command; if none is supplied, the default settings are reestablished. If the list is terminated by a *return*, the current modes are changed. If the list is terminated by an *escape*, a temporary change is made that applies only to the following subcommand, as in the following example.

```
#Mode Instruction esc ##Type 01234cr
 01234: JUMP 8567
#
```

Modes are established for feedback (verbose or concise); output (bits, formatted, instruction, numeric, or text); addresses (absolute or symbolic); output line format (dense or expanded); and output radix (any base from 2 through 36).

The feedback modes control how debugger commands are reflected to the user: *concise* suppresses all "noise" feedback (such as command completion); *verbose* enables it. The output modes control the general representation of data: *bits* treats a datum as an unsigned magnitude; *formatted* treats it as a pattern of bits partitioned into contiguous fields according to a designated format (see Format command); *instruction* treats it as a machine instruction and disassembles it; *numeric* treats it as a signed value, if that is appropriate for the machine; and *text* treats it as a representation of a string of characters. The address modes control whether numeric-mode values are to be converted to symbols (if possible): *absolute* suppresses the symbol look-up; *symbolic* enables it. The line-format modes control the density of displays: *dense* suppresses most

debugger-generated line-feeds so as to show more information per line, *expanded* enables them.

When formatted output is selected, the name of the output format must be specified, as in:

```
#Mode Formatted F1 cr.
```

Output radix sets the number base for the representation of numeric data (note that numeric input data self-identify the number base). For example,

```
#Mode Radix 16 cr
```

causes current output radix to become hexadecimal.

Format. Permits the user to name and define a format as a list of fields, each of which is a designated number of bits wide. The field widths are supplied as a list of numeric constants (separated by commas or spaces).

```
#Format F1esc 2 4 6 8cr
#

#Mode Formatted F1 esc #Type 0cr
 00: 00,00,00,00   #
```

If the format command is terminated without having defined a format, all defined formats are displayed, as in

```
#Format cr
 F1 2,4,6,8   #
```

Comment. Following an initial semicolon, ignores all subsequent inputs up to and including a line terminator.

```
#; THIS IS A COMMENT--IT DOES NOT GET INTERPRETED.cr
#
```

New-symbol. Adds a list of new user symbols to the (possibly empty) global symbol table. Each new symbol in the list is supplied as a name followed by a *space* or an *escape* followed by an expression giving its location.

```
#New-symbols ?(((new-symbol) <ESC> (expression))-list)
#New-symbols PATCHesc <at> 070000cr
#Type PATCH,PATCH-1,PATCH+1cr
 PATCH: 00    067777: 00    PATCH+01: 00   #
```

Kill-symbol. Removes a list of user symbols from the open local or global symbol table.

```
#Kill-symbols ?(list-of-user-symbols)
#Kill-symbols PATCHcr
#Type 067777:+2cr
 067777: 00    070000: 00    070001: 00   #
```

Open-symbol-table. Opens a local (program-specific) symbol table if one is specified; the currently open local symbol table, if any, is closed in any case. After this command is executed, the available symbols include the global symbols plus the local symbols of the specified program; if no program is specified, only the global symbols are available.

```
#Open-program-symbols ?(program-name) or not <close the open local symbol table>
#Open-program-symbols cr
#
```

Execution Control

Execution control commands provide for user control over execution of the target program. They permit the user to continue execution, transfer to a designated location, set and clear breakpoints or edit break-time programs, and single-step the target program. The execution control commands are

Go. Passes control to the target machine in its current state. If an argument is supplied, its value is first stored into the program counter. The argument can be an arbitrary expression, so long as it evaluates to a legal memory address.

```
#Go (to) ?(expression) or empty
#Go (to) 01000cr
```

Break. Displays or sets breakpoints in the target machine. The two classes of breakpoints are known as event breakpoints and reference breakpoints. There is a fixed set of event breakpoints defined for any given target machine; each describes a type of event whose occurrence causes the emulator to break if the corresponding event breakpoint is set. The set of event breakpoints always includes (1) every instruction-execution (single step), (2) every branch of control, and (3) every memory write; other events are defined for each machine as appropriate. Reference breakpoints cause the emulator to break when a specific type (read, write, and/or execute) of reference to a specific location occurs. Reference breakpoints may always be set on memory locations; other spaces in which reference breakpoints may be set are detailed in the tool-specific user guide. Any number of reference breakpoints may be set at any time.

The break command followed immediately by a *return* causes all existing breakpoints (*i.e.*, those in the breakpoint data base) to be displayed; if a break-time program is associated with a breakpoint, its number is also displayed. Otherwise, a list of either events or ranges (reference locations) for the setting of breakpoints is supplied. If a list of ranges has been entered and terminated with an *escape*, then a list of read, write, or execute reference-break conditions is specified next (as permitted at those locations); the default is all three types. Whenever a breakpoint is set for an event or a location, any earlier breakpoint for that same event or location is superseded.

If the list of events or break types is terminated by an *escape*, as in the second example below, a break-time "program" may be supplied to be executed by the debugger when the break is encountered. The following commands are permitted within such a break program: Clear, Comment, Debreak, Evaluate, Go, If, Jump-history, Locate, Mode, Open, Set, Type, and Use. Replacement within a locate or type command is not permitted in a break-time program. Any number of commands can be included in a break program; the program is terminated by an empty command (terminator only).

```
#Break (at) ?(event-list) or ((expression-range)-list) or <RETURN>
<? for list of events>
#Break (at) 0123:0456, 0712esc (after doing) cr
<R,W,X>  #

#Break (at) 01000esc (after doing) Xecute esc
 ##Type 0cr
 ##Go (to) cr
 ##cr
<Program number is [1]>  #

#Break (at) TICKcr
#
#Break (at) cr
0123-0456 <R,W,X>  0712 <R,W,X>  01000 <X>[1]  TICK <event>  #
```

During program execution, if an event break is detected, or if a reference break (read, write, or execute) is detected at a location for which the corresponding break type has been specified, then execution is terminated before beginning the next target machine cycle and control passes to the debugger to process the break. If a break-time program has been supplied for that break event or location, the program's commands are executed in order by the debugger until either a go command or the end of the program is encountered. If several breaks occur on the same cycle, the program associated with each of them is executed; the order of break-program execution corresponds to the order in which the breaks are reported by the emulator. If every break causes execution of a Go command, then the target program is automatically resumed, provided there is no ambiguity as to where execution is to resume. Otherwise (*i.e.*, if any break had no program or failed to execute a Go command), a message describing each of the breaks is displayed and the normal command level of the debugger is entered.

Debreak. Clears event breakpoints or reference breakpoints at locations in the target machine. The default is to clear all breakpoints. Examples of debreak commands are

```
#Debreak (from) 0234:+4cr
#Break (at) cr
0123-0233 <R,W,X>  0241-0456 <R,W,X>  0712 <R,W,X>  01000 <X>[1]
TICK <event>  #

#Debreak (from) esc all (confirm)cr
#Break (at) cr
#
```

Program-edit. Displays a designated break-time program or permits it to be edited. A program number must be designated that corresponds to an existing break-time program. Program numbers are shown when the breakpoint data base is displayed (see the break command). If the command is terminated by a *return*, the entire program is displayed; if by an *escape*, the program is displayed line by line for editing.

```
#Break (at) STEPesc
 ##Type @OLDCCcr
 ##Go (to) cr
 ##cr
<Program number is [2]>  #Break (at) cr
0123-0233 <R,W,X>  0241-0456 <R,W,X>  0712 <R,W,X>  01000 <X>[1]
TICK <event>  STEP <event>[2]
#Program-edit ?(program-number) (<ESC>-to-edit or <RETURN>-to-view)
#Program-edit 2cr
   Type @OLDCC
   Go (to)
#
```

When editing a line of a break-time program, the user can specify that the next (\) or prior (↑) line be displayed or that a replacement (R) of the current line or an insertion (I) in front of the current line be made. Editing is terminated by an empty editing specification. Replacement or insertion is identical to the specification of a break-time program within the break command in that a subcommand mode is entered where successive break-time commands can be entered until an empty command is supplied; then editing continues with the next line of the program. An extra (dummy) last line is added when editing a program so that new commands can be inserted at the end; the dummy line is discarded when the command is terminated.

```
#Program-edit 2esc
   Type @OLDCC   ;?(↑ <prior>) or (\ <next>) or ((I<nsert>) or (R<eplace>)
 (commands))
   Type @OLDCC   ;Replace
 ##Mode Instruction esc ###Type @OLDCCcr
 ##cr
   Go (to)   ;cr
#Program-edit 2cr
   Mode Instruction  ###Type @OLDCC
   Go (to)
#
```

Single-step. Transfers control to the target program through the program counter for execution of one instruction. The single coded character *line-feed* effects this command.

Display

The display commands permit the user to search or examine the contents of designated locations (and, in two cases, optionally permit their replacement) or to evaluate expressions. The commands are:

Type. Displays location and contents of a list of expression-ranges, permitting the contents of each location to be replaced if the list is terminated by an *escape*, as in the following example.

```
#Type ?((expression-range)-list) optional-<escape>-to-modify
#Type 0:2esc    00: 00 = 1cr
 01: 00 = 2cr
 02: 00 = 3cr
#
```

The replacement value can actually be a list of expressions, the values of the expression

in the list going into successive locations starting with the one last displayed. If no new value is supplied before the terminator, the existing value is not modified.

```
#Type 0:2esc     00: 01 = 2esc     01: 02 = esc     02: 03 = 1esc     #
```

In Display-with-replacement only, the coded characters *back-slash* and *up-arrow* can also serve as terminators and perform special functions: *back-slash* causes the next location to be displayed for replacement and *up-arrow* causes the prior location to be displayed for replacement; both of these terminator characters permit the user to step beyond the limits of the ranges entered as arguments to the Type command.

```
#Type 010esc     010: 00 = 1↑     07: 00 = 2\     010: 01 = 3\     011: 00 = ↑
 010: 03 = 4↑     07: 02 = 5↑     06: 00 = \     07: 05 = \     010: 04 = \
 011: 00 = 6\     012: 00 = 7cr
#
```

The last location displayed by a type command becomes the "open" location, and the location following the last one displayed or replaced becomes the "next" location (see the next four commands).

Same. Redisplays the "open" location (see the Type command). The single coded character ":" effects this command. The commands Same, Prior, and Next are all shown in the following example.

```
#:  02: 01   #↑  01: 02   #\  02: 01   #\  03: 00   #
```

Prior. Displays the location at one less than the "open" location (see the Type command). The single coded character *up-arrow* effects this command. See the examples under Type, Same, and Equals.

Next. Displays the "next" location (See the Type command; the mode in which the open location was last displayed determined how far it was advanced to the "next" locations.) The single coded character *back-slash* effects this command. See the examples under Type, Same, and Equals.

Equals. Displays the "open" location (see the Type command) as bits or as a number if the current output mode is already bits. The single coded character "=" effects this command. In the following example format *F2* has been declared consisting of four half-word fields.

```
#Mode Formatted F2cr
#:  010: 00,01,02,03   #=  010: 01   #\  011: 02,03,04,05   #\  013: 06,07,00,01
#↑  012: 04,05,06,07
```

Locate. Finds cells in a list of expression-ranges that contain (or do not contain) a specified value, examining only those bits designated by an optional mask, and displays their locations and contents, permitting each displayed value to be replaced if the list is terminated by an *escape*. The comparison value and mask are expressions terminated by an *escape*; the comparison value defaults to "NON 0" and the mask defaults to all 1's. The search is performed over a list of ranges, as for the Type command.

```
#Locate ?((expression) or NON (expression)) <match value defaults to NON 0>
#Locate NON 0esc (with mask) ?(optional-expression) <mask value>
#Locate NON 0 (with mask) esc<not zero> (in) ?((expression-range)-list)
 optional-<ESC> to-modify
#Locate NON 0 (with mask) <not zero> (in) 0:020cr
 00: 01    01: 02    02: 03    07: 05    010: 04    011: 06    012: 07
```

It is important that the comparison value, the mask, and the data be properly aligned. For example,

```
#Locate 070esc (with mask) 070esc (in) 0:31cr
```

displays all cells from 0 through 31 whose second octal digit from the right contains all 1's.

When the command is terminated by an *escape* the debugger stops after each display to permit replacement, as for the Type command.

```
#Locate escape<non-zero> (with mask) 07esc (in) 0:020esc    00: 01 = 3cr
 012: 07 = cr
#
```

Jump-history. Displays the most recent target-program jumps in the order they occurred. The number of such jumps to display (taken modulo the default value) may be supplied.

```
#Jump-history ?((expression) or (empty <all>))
#Jump-history 3cr
01000--0200(2 times)  0300--0100 #
```

Evaluate. Prints the value of a single expression. It has no effect on the open location and does not permit replacement.

```
#Non-symbols PATCHesc <at> 070000cr
#Evaluate PATCHesc - 070000  #
```

Storage

Storage commands change the contents of designated locations without displaying them and without changing the "open" location. The storage commands are

Clear. Clears the contents of a list of expression-ranges to all zero bits. Clearing an event for which a breakpoint has been established causes the event to be deactivated; it may be reactivated with a Set command. This may be of benefit when a break-time program has been associated with the event as the breakpoint data-base entry for that event is not affected.

```
#Clear 0.3esc  #
```

Set. Sets the contents of a list of expression-ranges to the value of an expression or (on default) to all one-bits. If the list is terminated by an *escape*, a single replacement expression is accepted; if it is terminated by a *return*, the default value of all 1's is used. The replacement expression is truncated to fit into the designated locations, if necessary. Setting an event for which a breakpoint has not been established (*i.e.*, for which there is no entry in the breakpoint data base) causes the event to be activated for a single occurrence of that event (with no break program associated), after which the event is automatically cleared.

```
#Set ?((expression-range)-list)
#Set 0.3cr

#Set 0.3esc - 2cr
#
```

TARGET EXECUTION STATE

Target execution is initiated, or resumed, through explicit commands (exec Go, debugger Go or Single-step). Execution proceeds until a terminating event occurs, causing control to return to the appropriate PRIM command level. When execution terminates, the entire emulated context -- including clocks and outstanding IO operations -- is cleanly frozen until the next time execution is resumed. Except for explicit modifications to the context made by the user at the command level, the termination and subsequent resumption of execution is transparent to the target machine. The terminating events are

The target machine halts normally or is interrupted (by the emulator) due to the occurrence of some anomaly condition. A message to that effect is generated. The anomalies being monitored are listed in the tool-specific user guide.

The user enters an *abort*. The abort character is echoed and, after execution is stopped, a status message is output indicating the point of interruption.

The emulator detects the occurrence of a break condition established by the user via the debugger breakpoint command. The establishment of breakpoints and the subsequent interruption of execution at the time of their occurrence is the primary program debugging tool in PRIM.

An IO error occurs. A message detailing the particular device involved and the nature of the error is output. IO errors always return control to the exec state; the error messages and their meanings are listed at the end of this section.

When one of these conditions occurs, it is logged and execution continues until the end of the current cycle of the target emulator. It is therefore possible for multiple conditions to result in a single stop. When this is the case, the action and message appropriate to each of the conditions is produced.

When a breakpoint is detected, the debug program, if any, associated with each breakpoint is executed by the debugger before control returns to the command level. Should some break program terminate without a Go -- or should there be some break with no break program -- a message describing the break is output and the command level is entered. Otherwise, execution is automatically resumed; the user receives no indication that a breakpoint occurred unless the break program itself produced output.

TARGET I/O

The target machine that runs in PRIM consists of a processor (CPU) in some particular configuration built by the user to resemble the actual configuration required by his programs. A configuration is built -- before execution is begun -- by installing peripheral devices and establishing values for various machine options (see the exec Install and Set commands). After an emulated device has been installed, and before IO operations can proceed on that device, a (TENEX) file or assignable device must be associated with that emulated device (see the exec Mount command). Subsequent IO operations addressed to that device are then performed on the mounted file.

A mounted file may contain either direct device data (binary) or ASCII text; in the latter case, characters are translated between ASCII and the actual device character set as

they are processed. (If the device character set does not include lower case, input lower case letters are converted to upper case before translation.) When the target device is a record-oriented device (*e.g.*, card reader or punch) and the file is ASCII, then each record operation is performed on a line of the ASCII text file, including truncation and/or blank padding on input.

The mount option *THIS-TERMINAL* associates the user terminal (the one being used to communicate with PRIM) with a given device. When the terminal has been mounted on some device, then input from the terminal is switched between PRIM and the target machine every time execution is resumed and terminated. The intervention characters, however, retain their intervention meanings. To allow the full ASCII character set to be input to the target device from the terminal, there is a *control-shift* escape character defined during target execution. To help distinguish PRIM output from target output directed to *THIS-TERMINAL*, all PRIM-generated output is prefixed with the herald "--> " at the beginning of a new line. This applies in particular to both stopping messages and typeout resulting from break-time debugger programs.

I/O ERROR MESSAGES

Various I/O errors may occur. When any one occurs, execution -- including the error-generating operation -- is suspended, and control returns to the PRIM exec. When execution is next resumed, the suspended operation is retried unless it has been explicitly canceled by the user using the exec Cancel command.

"File not mounted."

The indicated device has no file mounted. If a file is mounted before execution is next resumed, the operation will be performed then. (An installed device to which no IO is directed need not have a mounted file in order to run.) The operation may instead be canceled.

This message is also produced when an output operation occurs on a device which has been mounted for input only, and vice versa. Again, a second file must be mounted on the appropriate side of the device in order to proceed normally with the program.

"File not open."

The indicated device has an inaccessible file mounted on it. The device must either be reassigned or unmounted and then mounted. The situation is similar to the case above, except for the possibility of reassigning.

"Improper tape format detected."

TENEX files which are mounted on target magnetic tape devices are encoded in a unique internal format that requires such files to be used only for PRIM magnetic tape devices. The mounted file is inconsistent with that format. The device must be unmounted and replaced with a proper tape file.

"Device not installed."

A device that is referenced by the program is not installed. Should the missing device be required, there is no way to continue this session, since device installation is no longer allowed. Should the reference be a mistake, execution may be continued down a different path (the operation will be automatically canceled when execution resumes).

"ASCII input character not recognized -- ignored."

The last character read from the ASCII input file on the designated device was not translatable into the character set of the device. The character has been skipped over; resuming execution causes the read operation to continue with the next character in the file. The position of the offending character in the file may be determined via the exec Filestatus command, specifying the indicated device.

Any other error indicates a bug either in the emulator or in PRIM. Such errors should be reported.